REPORT DOCUMENTATION PAGE

AFRL-SR-AR-TR-03-

0431

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYP
	30 September 2003	Final Report for 1 Sept 2002--30 Sept 2003

4. TITLE AND SUBTITLE
Fast, Robust, Real-Time Trajectory Generation for Autonomous and Semi-Autonomous Nonlinear Flight Systems

5. FUNDING NUMBERS
F49620-02-C-0094

6. AUTHOR(S)
Michael L. Larsen
Randal W. Beard
Timothy McLain

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

INFORMATION SYSTEMS LABORATORIES, INC.
10070 Barnes Canyon Road.
San Diego, CA

BRIGHAM YOUNG UNIVERSITY
Provo, Utah 84602

8. PERFORMING ORGANIZATION REPORT NUMBER
3507-FINAL

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

USAF, AFRL
AF OFFICE OF SCIENTIFIC RESEARCH
4015 WILSON BLVD ROOM 713
ARLINGTON VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

20031028 116

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Report developed under STTR contract for topic AF02T002. A path planner and trajectory generator for real time autonomous flight control was developed which is capable of generating extremely complicated paths that account for pop-up and dynamically changing threats. A feasible hierarchal decomposition of the problem using Voronoi diagrams, which affords a significant reduction in the search space, is reported. The trajectory generation problem is decomposed into three distinct, but tightly coupled pieces: waypoint path planning (WPP), dynamic trajectory smoothing (DTS), and adaptive trajectory tracking (ATT). The essential idea is to give the trajectory generator a similar mathematical structure as the physical vehicle. The approach uses a simple algorithm to generate smoothed trajectories in real-time without performing any on-line optimization. The trajectories that are generated by the DTS have the same path length as the waypoint path generated and also minimize the deviation from the waypoint path. The algorithms developed are suitable for real-time path generation for moving threats, terrain obstructions, and weather. Potential applications include small UAVs and smart munitions systems.

14. SUBJECT TERMS
STTR Report, UAV, trajectory generation, way point path planning

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

Real-Time Trajectory Generation for Autonomous Nonlinear Flight Systems

AF02T002 Phase I STTR Final Report
Contract # F49620-02-C-0094

Principal Investigators

Michael Larsen
Information Systems Laboratory Inc.
10070 Barnes Canyon Road
San Diego, CA 92121
voice: (858) 535-9680
fax: (858) 535-9848
email: mlarsen@islinc.com

Randal W. Beard
Department of Electrical and Computer Engineering
Brigham Young University
Provo, Utah 84604 USA
voice: (801) 422-8392
fax: (801) 422-0201
email: beard@ee.byu.edu

Timothy W. McLain
Department of Mechanical Engineering
Brigham Young University
Provo, Utah 84604 USA
voice: (801) 422-6537
email: tmclain@et.byu.edu

Executive Summary

Unmanned aerial vehicle and smart munition systems require robust, real-time path generation to account for terrain obstructions, weather, and moving threats such as radar, jammers, and unfriendly aircraft. The increasing speed and power of computational resources has enabled the development of autonomous flight control systems which are capable of dealing with the complex task of path planning in dynamic and uncertain environments. In Phase I a feasible, hierarchal approach for real-time motion planning of autonomous vehicles was developed.

Our approach decomposes the trajectory generation problem into three tasks to be performed by the UAV: waypoint path planning (WPP), dynamic trajectory smoothing (DTS), and adaptive trajectory tracking (ATT). The WPP plans paths at a high level without regard for the dynamic constraints of the vehicle. This affords a significant reduction in the search space, enabling the generation of extremely complicated paths that account for pop-up and dynamically changing threats. The essential idea of the DTS is to give the trajectory generator a similar mathematical structure as the physical vehicle. The DTS uses a simple, but novel algorithm to generate smoothed trajectories in real-time without performing any on-line optimization. The trajectories that are generated by the DTS have the same path length as the waypoint path generated by the WPP and also minimize the deviation from the waypoint path. The third step uses an adaptive nonlinear control technique, backstepping, to transform the trajectory generated by the DTS to a feasible trajectory that can be followed by an autopilot with appropriate velocity, altitude and heading commands.

The benefits of the approach are:

- The algorithms are computationally efficient and can handle hundreds of threats, including pop-up threats.
- The algorithms do not require any on-line optimization.
- The approach is well suited for applications with timing constraints. Planning can take place at the waypoint level, where it is trivial to calculate path length, and therefore the estimated time-of-arrival (ETA). The DTS and ATT then guarantee that the ETA remains unchanged. The WPP and DTS have been used successfully in coordinated control problems where the coordination objective is to sequence the ETA of different vehicles.[1]
- The trajectories are represented in a compact fashion, in both space and time. In particular, this allows higher-level task planning algorithms to reason about the feasibility, or desirability of different trajectories.

Contents

1 Technical Objectives of Phase I

The overall objectives of Phase I were to to develop concept algorithms for real-time trajectory generation of autonomous nonlinear flight systems and determine the system requirements for their implementation. Specifically:

- Develop robust algorithms for a hierarchal trajectory generator which enables a UAV to plan its path in response to static and dynamic threats.
- Determine the real-time computational requirement for such algorithms, as well as required failure management and redundancy.

2 Introduction

The increasing power of computational resources makes possible the development of autonomous flight control systems which are capable of dealing with the complex task of path planning in dynamic and uncertain environments. Unmanned aerial vehicles (UAVs) require robust, real-time path generation to account for terrain obstructions, weather, and moving threats such as radar, jammers, and unfriendly aircraft. Such path planning algorithms and route navigation aids are needed to accomplish envisioned future UAV missions.[2]

There are two general approaches to trajectory generation: interpolation of a trajectory database and formulation of the trajectory as the solution to an optimal control problem. Methods which query and interpolate trajectory databases fall into several categories: probabilistic roadmaps,[3] lazy probabilistic roadmaps,[4] and rapidly-exploring trees.[5,6] Probabilistic roadmaps are path-planning algorithms which consist of off-line building of a graph of uniformly spaced randomly selected configurations called milestones. A recent extension[7] of the probabilistic roadmap approach uses Lyapunov function scheduling to deal with system dynamics in an environment with moving obstacles. For aerospace systems with complex high-dimensional dynamics, this motion planning approach is based on a quantization of system dynamics into a library of feasible trajectory primitives.[8]

Trajectory generation via numerical solution of optimal control problems[9–12] is computationally intensive and requires recently developed techniques from geometric nonlinear control theory for feasible implementation. These techniques are based on finding a differentially flat output for the system.[13–15] From such an output and its derivatives, the complete differential behavior of the system can by reconstructed. In Ref. 16, 17, a flat output is used to find a lower dimensional space in which trajectory curves are generated using B-splines and sequential quadratic programming. In a similar vein, Refs. 18, 19 transform the nonlinear optimization problem to a linear one using feedback linearization, which requires finding a flat output, making it possible to convert constrained dynamic optimization problems into unconstrained ones. In Ref. 20, the differential flatness property was used to develop an iterative approach to finding a

feasible solution which satisfies terminal path constraints using an H_∞ estimator.

The path planner developed in Phase I uses a modified Voronoi diagram[21] to generate possible paths. The Voronoi diagram is then searched via Eppstein's k-best paths algorithm.[22] Similar path planners have been previously reported in Refs. 23–25. The basic idea is to plan a polygonal path through a set of threats using a Voronoi algorithm in connection with an A* or Dijkstra algorithm. The polygonal paths are then made flyable by a trajectory smoother that dynamically smooths through the corners of the paths such that the curvature of the smoothed path is flyable by the UAV. The described algorithm works well when the minimum turning radius is small compared to the path links.

3 Trajectory Generation Algorithm Development

The overall architecture is built around a systematic division of the problem into five distinct hierarchical layers: path planning, trajectory smoothing, trajectory tracking, autopilot, and the UAV. In this paper, paths refer to a series of waypoints which are *not* time-stamped, while trajectories will refer to time-stamped curves which specify the desired inertial location of the UAV at a specified time. Figure 1 shows a schematic of the architecture. At the top level is a path plan-

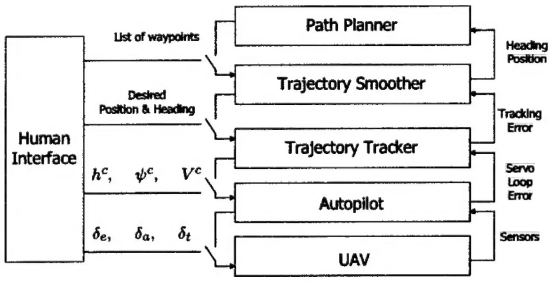


Figure 1: Proposed system architecture.

ner (PP). It is assumed that the PP knows the location of the UAV, the target, and the location of a set of threats. The PP generates a path

$$\mathcal{P} = \{v, \{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_N\}\},$$

where $v \in [v_{\min}, v_{\max}]$ is a feasible velocity and $\{\mathbf{w}_i\}$ is a series of waypoints which define straight-line segments along which the UAV attempts to fly. Note that at the PP level, the path is compactly represented by $\mathcal{P}$. Higher level decision making algorithms reason and make decisions according to the data represented in $\mathcal{P}$.

The Trajectory Smoother (TS) transforms, in real-time, the waypoint trajectory into a time parameterized curve which defines the desired inertial position

of the UAV at each time instant. The output of the TS is the desired trajectory $\mathbf{z}^d(t) = (z_x^d(t), z_y^d(t))^T$ at time t.

The Trajectory Tracker (TT) transforms $\mathbf{z}^d(t)$ into desired velocity command V^d, altitude command h^d, and heading command ψ^d. The autopilot receives these commands and controls the elevator, δ_e, and aileron, δ_a, deflections and the throttle setting δ_t.

Recognizing that it will be useful for human operators to interact with the UAV at different autonomy levels, careful attention has been given to the human interface. As shown in Figure 1, the human can interact with the UAV at the stick-and-throttle level, the autopilot command level, the time-parameterized trajectory level, or at the waypoint path planning level.

3.1 Trajectory Tracker

This section provides a description of the trajectory tracker. A complete description is contained in Refs. 26, 27. We will assume that the UAV/autopilot combination is adequately modeled by the kinematic equations

$$\begin{aligned} \dot{x} &= v^c \cos(\psi) \\ \dot{y} &= v^c \sin(\psi) \\ \dot{\psi} &= \omega^c, \end{aligned} \tag{1}$$

where (x, y) is the inertial position of the UAV, ψ is its heading, v^c is the commanded linear speed, and ω^c is the commanded heading rate. The dynamics of the UAV impose input constraints of the form

$$\begin{aligned} 0 < v_{min} \leq & v^c \leq v_{max} \\ -\omega_{max} \leq & \omega^c \leq \omega_{max}. \end{aligned} \tag{2}$$

As we will describe in the next section, the trajectory generator produces a reference trajectory that satisfies

$$\begin{aligned} \dot{x}_r &= v_r \cos(\psi_r) \\ \dot{y}_r &= v_r \sin(\psi_r) \\ \dot{\psi}_r &= \omega_r \end{aligned} \tag{3}$$

under the constraints that v_r and ω_r are piecewise continuous and satisfy the constraints

$$\begin{aligned} v_{min} + \epsilon_v \leq & v_r \leq v_{max} - \epsilon_v \\ -\omega_{max} + \epsilon_\omega \leq & \omega_r \leq \omega_{max} - \epsilon_\omega, \end{aligned} \tag{4}$$

where ϵ_v and ϵ_ω are positive control parameters.

The trajectory tracking problem is complicated by the nonholonomic nature of Equations (1) and the input constraint on the commanded speed v^c.

The first step in the design of the trajectory tracker is to transform the tracking errors to the UAV body frame as follows:

$$\begin{bmatrix} x_e \\ y_e \\ \psi_e \end{bmatrix} = \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r - x \\ y_r - y \\ \psi_r - \psi \end{bmatrix}. \tag{5}$$

Accordingly, the tracking error model can be represented as

$$\begin{aligned} \dot{x}_e &= \omega^c y_e - v^c + v_r \cos(\psi_e) \\ \dot{y}_e &= -\omega^c x_e + v_r \sin(\psi_e) \\ \dot{\psi} &= \omega_r - \omega^c. \end{aligned} \tag{6}$$

Following Ref. 28, Eq. (6) can be simplified to

$$\begin{aligned} \dot{x}_0 &= u_0 \\ \dot{x}_1 &= (\omega_r - u_0)x_2 + v_r \sin(x_0) \\ \dot{x}_2 &= -(\omega_r - u_0)x_1 + u_1, \end{aligned} \tag{7}$$

where

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \psi_e \\ y_e \\ -x_e \end{bmatrix} \tag{8}$$

and

$$\begin{bmatrix} u_0 \\ u_1 \end{bmatrix} = \begin{bmatrix} \omega_r - \omega^c \\ v^c - v_r \cos(x_0) \end{bmatrix}.$$

The input constraints under the transformation become

$$\begin{aligned} -\epsilon_\omega \leq u_0 \leq \epsilon_\omega \\ v_{min} - v_r \cos(x_0) \leq u_1 \leq v_{max} - v_r \cos(x_0). \end{aligned} \tag{9}$$

Obviously, Eqs. (5) and (8) are invertible transformations, which means $(x_0, x_1, x_2) = (0, 0, 0)$ is equivalent to $(x_e, y_e, \psi_e) = (0, 0, 0)$ and $(x_r, y_r, \psi_r) = (x, y, \psi)$. Therefore, the original tracking control objective is converted to a stabilization objective, that is, our goal is to find feasible control inputs u_0 and u_1 to stabilize x_0, x_1, and x_2.

Note from Eq. (7) that when both x_0 and x_2 go to zero, that x_1 becomes uncontrollable. To avoid this situation we introduce another change of variables. Let

$$\bar{x}_0 = m x_0 + \frac{x_1}{\pi_1}, \tag{10}$$

where $m > 0$ and $\pi_1 \triangleq \sqrt{x_1^2 + x_2^2 + 1}$. Accordingly, $x_0 = \frac{\bar{x}_0}{m} - \frac{x_1}{m\pi_1}$. Obviously, $(\bar{x}_0, x_1, x_2) = (0, 0, 0)$ is equivalent to $(x_0, x_1, x_2) = (0, 0, 0)$. Therefore it is

sufficient to find control inputs u_0 and u_1 to stabilize $\bar{x}_0$, x_1, and x_2. With the same input constraints (9), Eq. (7) can be rewritten as

$$\begin{aligned}
\dot{\bar{x}}_0 &= (m - \frac{x_2}{\pi_1})u_0 + \frac{x_2}{\pi_1}\omega_r \\
&\quad + \frac{1+x_2^2}{\pi_1^3} v_r \sin\left(\frac{\bar{x}_0}{m} - \frac{x_1}{m\pi_1}\right) - \frac{x_1 x_2}{\pi_1^3} u_1 \\
\dot{x}_1 &= (\omega_r - u_0)x_2 + v_r \sin(\frac{\bar{x}_0}{m} - \frac{x_1}{m\pi_1}) \\
\dot{x}_2 &= -(\omega_r - u_0)x_1 + u_1.
\end{aligned} \tag{11}$$

In Refs. 26,27 we have shown that if

$$u_0 = \begin{cases} -\eta_0 \bar{x}_0, & |\eta_0 \bar{x}_0| \le \epsilon_\omega \\ -\text{sign}(\bar{x}_0)\epsilon_\omega, & |\eta_0 \bar{x}_0| > \epsilon_\omega \end{cases} \tag{12}$$

$$u_1 = \begin{cases} \underline{v}, & -\eta_1 x_2 < \underline{v} \\ -\eta_1 x_2, & \underline{v} \le -\eta_1 x_2 \le \bar{v} \\ \bar{v}, & -\eta_1 x_2 > \bar{v} \end{cases}, \tag{13}$$

where $\underline{v} = v_{\min} - v_r \cos\left(\frac{\bar{x}_0}{m} - \frac{x_1}{m\pi_1}\right)$ and $\bar{v} = v_{\max} - v_r \cos\left(\frac{\bar{x}_0}{m} - \frac{x_1}{m\pi_1}\right)$, and η_0 and η_1 are sufficiently large (made precise in Refs. 26,27), then the tracking error goes to zero asymptotically.

Note the computational simplicity of Equations (12)-(13). We have found that the tracker can be efficiently implemented on the autopilot hardware discussed in Section 3.4.3.

Figure 2 shows a reference trajectory of the UAV in green and the actual UAV trajectory in blue. Figure 3 plots the tracking errors verses time and demonstrates the asymptotically stable characteristics of the trajectory tracker.

3.2 Trajectory Smoother

Given a path $\mathcal{P}$, the objective of the Dynamic Trajectory Smoother (DTS) is to generate time-parameterized trajectories that are feasible within the dynamic constraints of the UAV. The essential idea is to use a nonlinear filter that has a mathematical structure similar to the dynamic structure of the vehicle to generate trajectories that smooth through the waypoints in real-time, while the vehicle traverses the trajectory.

The Trajectory Smoother (TS) translates a straight-line path into a feasible trajectory for a UAV with velocity and heading rate constraints. Our particular implementation of trajectory smoothing also has some nice theoretical properties including time-optimal waypoint following.[29,30]

We start by assuming that an auto-piloted UAV is modeled by the kinematics equations given in Eq. (1), with the associated constraints given in Eq. (2).

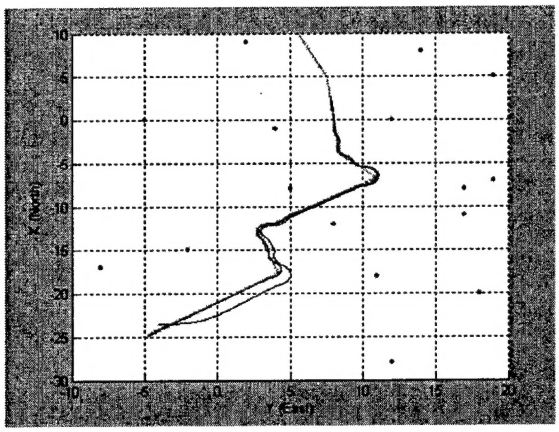


Figure 2: The simulation scenario: waypoint path (green), smoothed reference trajectory (red), and actual trajectory (blue).

The fundamental idea behind feasible, time-extremal trajectory generation is to impose on TS a mathematical structure similar to the kinematics of the UAV. The structure of the TS is given in Eq. (3) with constraints given by Eq. (4). To simplify notation, let $c = \omega_{\max} - \epsilon_\omega$.

With the velocity fixed at $\hat{v}$, the minimum turn radius is defined as $R = \hat{v}/c$. The idea of a minimum turn radius allows us to visualize the area of space that the UAV can reach in the next instant of time, i.e. the local reachability region.

With this in mind, it seems natural that for a trajectory to be time-optimal, it will be a sequence of straight-line path segments combined with arcs along the minimum radius circles (i.e. along the edges of the local reachability region). In fact, Anderson proved in Ref. 31 that this is the case. By postulating that ω_r follows a bang-bang control strategy during transitions from one path segment to the next, he showed that a κ-trajectory is time-extremal, where a κ-trajectory is defined as follows. As shown in Figure 5, a κ-trajectory is defined as the trajectory that is constructed by following the line segment $\overline{\mathbf{w}_{i-1}\mathbf{w}_i}$ until intersecting $\mathcal{C}_i$, which is followed until $\mathcal{C}_{\mathbf{p}(\kappa)}$ is intersected, which is followed until intersecting $\mathcal{C}_{i+1}$ which is followed until the line segment $\overline{\mathbf{w}_i\mathbf{w}_{i+1}}$ is intersected.

Note that different values of κ can be selected to satisfy different requirements. For example, κ can be chosen to guarantee that the UAV explicitly passes over each waypoint, or κ can be found by a simple bi-section search to make the trajectory have the same length as the original straight-line path,[31] thus facilitating timing-critical trajectory generation problems.

The TS implements κ-trajectories to follow waypoint paths. In evaluating the real-time nature of the TS, we chose to require trajectories to have equal path length as the initial straight line paths. The computational complexity to

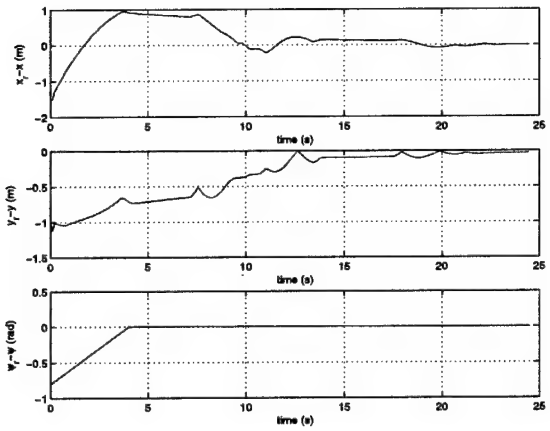


Figure 3: The trajectory tracking errors expressed in the inertial frame.

find ω_r is dominated by finding circle-line and circle-circle intersections. Since the TS also propagates the state of the system in response to the input ω_r, Eq. (3) must be solved in real-time. This is done via a forth-order Runge-Kutta algorithm.[32] In this manner, the output of the TS corresponds in time to the evolution of the UAV dynamics and ensures a time-optimal trajectory.

Figure 6 shows the essential idea of the DTS algorithm which was recently developed in.[31] Let $\mathbf{p}$ be a point on the bisector of the angle defined by the path segments $\ell_i = (\mathbf{w}_{i-1}, \mathbf{w}_i)$ and $\ell_{i+1} = (\mathbf{w}_i, \mathbf{w}_{i+1})$, and let $\mathcal{C}_p$ denote the circle of radius R centered at $\mathbf{p}$. The DTS tracks ℓ_i until $\mathcal{C}_L$ intersects $\mathcal{C}_p$ at point i_1, upon which u is set to $+c$. The DTS follows the circle defined by $\mathcal{C}_L$ until it reaches i_2, upon which it sets $u = -c$. The circle defined by $\mathcal{C}_p$ is followed until the DTS reaches i_3, upon which $u = +c$ until i_4 where $u = 0$. Using Pontrygin's minimum principle, it was shown in[31] that the trajectories generated by this algorithm are time-extremal trajectories. A novel aspect of this algorithm is that the point $\mathbf{p}$ can be adjusted to find the time-extremal trajectory whose path length is exactly equal to the corresponding path.[31] Let $\kappa \in [0, 1]$ be a dimensionless parameterization of the distance from the waypoint to an inscribed circle C whose center lies along the bisector of the angle. The

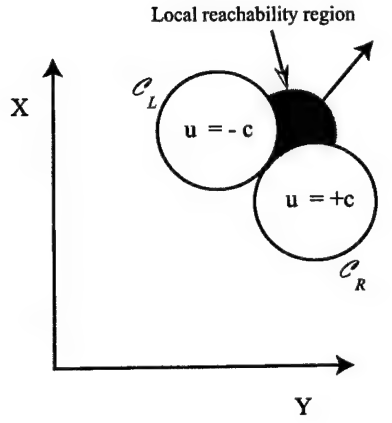


Figure 4: Local reachability region of the TS.

the value of κ corresponding to an equal length trajectory is given by

$$\Lambda(\kappa) \triangleq R\sqrt{4 - \left[1 + \sin\left(\frac{\beta}{2}\right) + \kappa\left(1 - \sin\left(\frac{\beta}{2}\right)\right)\right]^2} + R\left[1 + \kappa\left(\frac{1}{\sin\left(\frac{\beta}{2}\right)} - 1\right)\right]\cos\left(\frac{\beta}{2}\right) - R\left(\frac{\pi - \beta}{2} + 2\arccos\left(\frac{1 + \sin\left(\frac{\beta}{2}\right) + \kappa\left(1 - \sin\left(\frac{\beta}{2}\right)\right)}{2}\right)\right)$$

where $\beta \in [0, \pi)$ is the angle between the path segments ℓ_i and ℓ_{i+1} as shown in Figure 6. The function Λ can be shown to be monotonic, therefore its unique roots can be found quickly via a bisection search algorithm.

There are several advantages to the DTS approach. First, it is nicely matched to the WPP algorithms described in the previous section. Second, the approach has low computational overhead. In fact, trajectories are generated in real-time, as the vehicle moves. Third, it can be shown using optimal control techniques that the approach minimizes the time that the vehicle deviates from the Voronoi path. Finally, it can be adapted to allow low-level deconfliction, and to allow threat avoidance for dynamic threats. The algorithm described above can be modified using behavior-based approaches,[33] to allow local adjustments to the path in response to dynamically moving threats.

Hardware implementation of the TS has shown the real-time capability of this approach. On a 1.8 GHz Intel Pentium 4 processor, one iteration of the TS took on average 36 μ-seconds. At this speed, the TS could run at 25 KHz - well above the dynamic range of typical UAVs. Moving toward embedded systems,

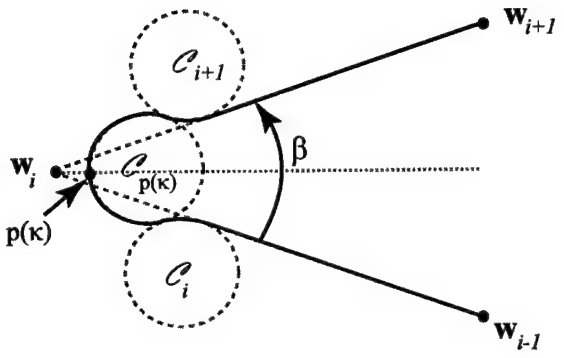


Figure 5: A dynamically feasible κ-trajectory.

we found that one iteration of the TS required a maximum of 47 milli-seconds on a Rabbit Microprocessor running at 29 MHz. The low computational demand allows the TS to be run in real-time at approximately 20 Hz on an embedded system on-board the UAVs.

3.3 Waypoint Path Planning

For many anticipated military and civil applications, the capability for a UAV to plan its route as it flies is important. Reconnaissance, exploration, and search and rescue missions all require the ability to respond to sensed information and to navigate on the fly.

In the flight control architecture shown in Figure 1, the coarsest level of route planning is carried out by the path planner (PP). Our waypoint planning technique centers around the construction and search of a Voronoi graph.[24] The Voronoi graph provides a method for creating waypoint paths through threats or obstacles in the airspace. A prime advantage of the Voronoi graph is the computational speed with which the graph can be created and searched.

In Phase I, we have modelled threats and obstacles in two different ways: as points to be avoided and as polygonal regions that cannot be entered. With threats specified as points, construction of the Voronoi graph is straightforward using existing algorithms. For an area with n point threats, the Voronoi graph will consist of n convex cells, each containing one threat. Every location within a cell is closer to its associated threat than to any other. By using threat locations to construct the graph, the resulting graph edges form a set of lines that are equidistant from the closest threats. In this way, the edges of the graph maximize the distance from the closest threats. Figure 7 shows an example of a Voronoi graph constructed from point threats.

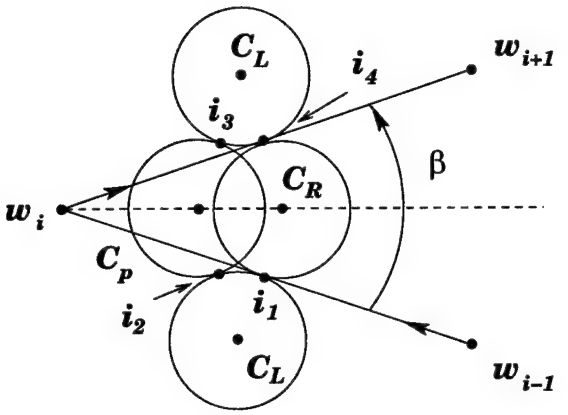


Figure 6: Graphical depiction of the essential idea behind the DTS.

Construction of a Voronoi graph for obstacles modelled as polygons is an extension of the point-threat method. In this case, the graph is constructed using the vertices of the polygons that form the periphery of the obstacles. This initial graph will have numerous edges that cross through the obstacles. To eliminate these infeasible edges, a pruning operation is performed. Using a line intersection algorithm, those edges that intersect the edges of the polygon are detected and removed from the graph. Figure 8 shows the initial polygon based graph and the final graph after pruning.

Finding good paths through the Voronoi graph requires the definition of a cost metric associated with travelling along each edge. In our work, two metrics have been employed: path length and threat exposure. A weighted sum of these two costs provides a means for finding safe, but reasonably short paths. Although the highest priority is usually threat avoidance, the length of the path must be considered to prevent safe, but meandering paths from being chosen.

Once a metric is defined, the graph is searched using an Eppstein search.[22] This is an computationally efficient search with the ability to return k best paths through the graph. Once k best paths are known, a coordination agent can choose which path to choose for each vehicle in the team (to ensure simultaneous arrival, for example). The points of this chosen path are passed on the the trajectory generator which smooths through the path, taking into account the dynamics and constraints of the vehicle.

Each edge of the Voronoi diagram is assigned two costs: a threat cost and a length cost. Threat costs are based on a UAV's exposure to enemy radar. Assuming that the UAV radar signature is uniform in all directions and is proportional to $1/d^4$ (where d is the distance from the UAV to the threat), the threat cost for travelling along an edge is inversely proportional to the distance (to the threat) to the fourth power. An exact threat cost calculation would involve the integration of the cost along each edge. An acceptable approximation

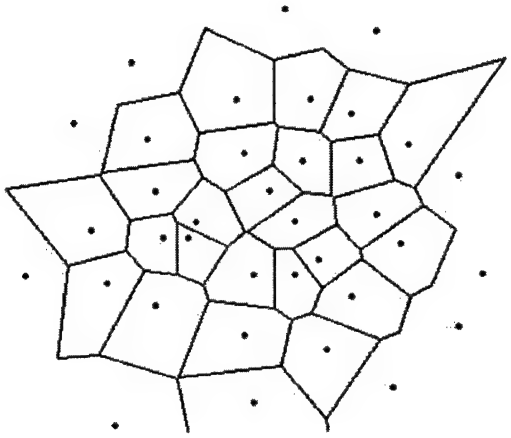


Figure 7: Voronoi graph with point threats

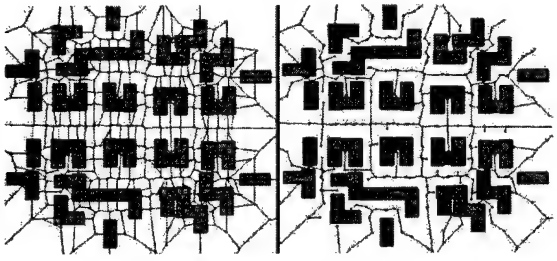


Figure 8: Voronoi graph before and after pruning with polygon threats

is to calculate the threat cost at several locations along an edge and take the length of the edge into account. In this work, the threat cost was calculated at three points along each edge: $L_i/6$, $L_i/2$, and $5L_i/6$, where L_i is the length of edge i. The threat cost associated with the ith edge is given by the expression

$$J_{threat,i} = L_i \sum_{j=1}^{N} \left(\frac{1}{d_{1/6,i,j}^4} + \frac{1}{d_{1/2,i,j}^4} + \frac{1}{d_{5/6,i,j}^4} \right), \tag{14}$$

where N is the total number of threats and $d_{1/2,i,j}$ is the distance from the 1/2 point on the ith edge to the jth threat. To include the path length as part of the cost, the length cost associated with each edge is

$$J_{length,i} = L_i. \tag{15}$$

The total cost for travelling along an edge comes from a weighted sum of the threat and length costs:

$$J_i = kJ_{length,i} + (1-k)J_{threat,i}. \tag{16}$$

The choice of k between 0 and 1 gives the designer flexibility to place weight on exposure to threats or fuel expenditure depending on the particular mission scenario. With the cost determined for each of the Voronoi edges, the Voronoi diagram is searched to find the lowest-cost path using Eppstein's algorithm.[22] Figure 2 shows the five best paths returned from a search of the Voronoi diagram.

Taken in different combinations, the edges of the Voronoi diagram provide a rich set of paths from the starting point to the target. A great advantage of the Voronoi diagram approach is that it reduces the path-planning problem from an infinite-dimensional search, to a finite-dimensional graph search. This important abstraction makes the path-planning problem feasible in real-time.

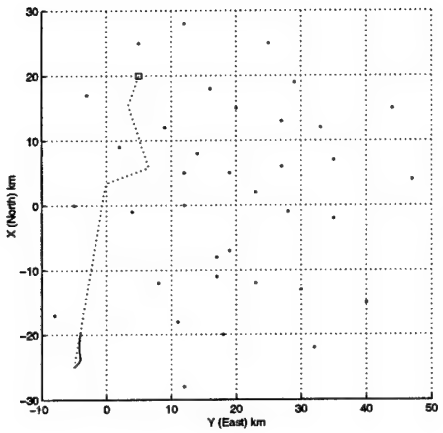
Voronoi diagram generation and search can be performed in a computationally efficient manner, generating a path in about 10 millisecond for battlefields with approximately 100 threats.

The problem, of course, is that straight-line paths are not feasible within the dynamic constraints of the UAV. The objective of the Dynamic Trajectory Smoother is to transform $\mathcal{P}$ into a dynamically feasible trajectory.

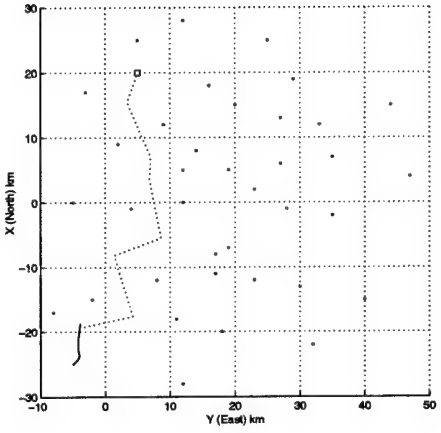
There are numerous applications where timing is critical. For example, it may be desirable that two or more UAVs strike a target simultaneously to maximize the element of surprise. The waypoint paths, together with a velocity $v \in [v_{\min}, v_{\max}]$ specify the time of completion. However, in smoothing the trajectory, if the path length is changed, the timing will be altered. Therefore if timing is critical, it is necessary to smooth the trajectories in such a way that path length is not altered.

3.3.1 Example

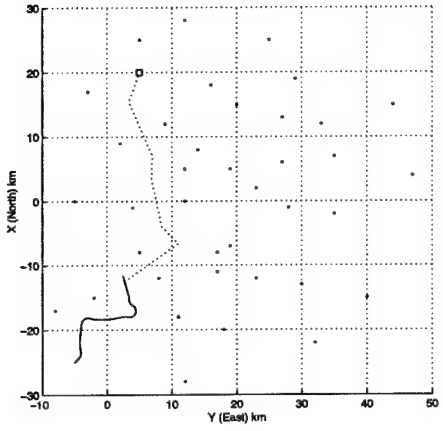
In this section we will illustrate the essential ideas via a Simulink simulation of the WPP/DTS combination. Figure 9 shows a battlefield scenario where a single UAV is assigned to visit several target locations. In the battlefield there are 36 known threat locations with several unknown threat locations that are detected by the UAV when it flies in the vicinity of the threat. We assume that the UAV is initially at position (-25 km, 5 km) and that the first target location, represented by a small box, is at (20 km, 5 km). The dotted line in Figure 9(a) is the initial waypoint path planned by the WPP. The solid line shows the trajectory produced by the DTS from the initial time until the instant immediately before the detection of the first pop-up threat. In Figure 9(b) a new threat at location (-15 km, -2 km) has been detected by the UAV. The new waypoint path is shown in the dotted line. Figure 9(c) shows the state of the WPP/DTS immediately after a second pop-up threat has been detected at position (-8 km,5 km). Finally Figure 9(d) shows the state of the WPP/DTS at the instant of time when the first target location has been reached. The WPP has planned a waypoint path, represented by the dotted line, to the second target which is represented by a square.



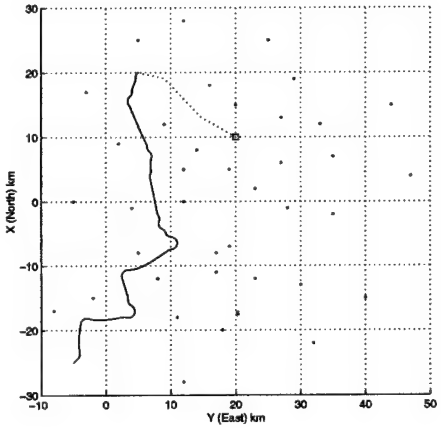
(a) First pop-up threat is about to be detected.



(b) The first pop-up threat is detected.



(c) The second pop-up threat is detected.



(d) The first target is reached.

Figure 9: Pop-up Scenario

3.4 Phase I Accomplishments

Real-time path planning algorithms were developed which generate trajectories are flyable by the UAV, computed in real-time, and can respond in real-time to pop-up threats and dynamically moving The trajectories can be represented in a compact fashion, in both space and time. In particular, this allows higher-level task planning algorithms to reason about the feasibility, or desirability of different trajectories.

3.4.1 Task 1: Algorithm Development

1. A way point path planner (WPP) was designed to accommodate dynamically moving threats. The waypoints move dynamically to guide the UAV to avoid moving threats. To overcome some of the limitations of the Voronoi approach to waypoint path planning, the path planner was modified to for account polygonal no-go regions and finite threat radius. The modified WPP creates a tree of path segments by connecting the current position of the UAV with the desired target via a straight line. If the line intersects with a threat, the original line is removed from the tree and four new lines are added that avoid the threat. These new lines are then searched recursively to ensure that they don't intersect with threats. When a specified number of paths exist from the start to the target (i.e. the graph is 'full enough') then the tree is searched for the shortest path.

   In a majority of cases, this approach is much faster than our initial Voronoi approach and produces a short, more intuitive path. Since the time to compute is small, it can be used in dynamic threat areas. The planning would simply be done at every time step with the threats extended in their respective directions of motion. In this way, the algorithm could plan with some sort of prediction.

   There are cases when this approach might fails to complete in a timely manner. When threats are very dense or the starting or target positions are enclosed in concave threat areas, then the algorithm may create too many paths to be searched efficiently. These issues can be addressed by grouping overlapping threats into convex hulls and identifing/removing areas where many edges exist.

   The path planner has been extended to handle polygon threat regions to account for no-fly zones and known threat areas of arbitrary shape. Improved error checking enables the planner to plan paths in previously problematic situations (i.e. not enough threats for full Voronoi graph). Recent Monte-Carlo simulations indicate this is feasible in real-time: for 30 8-sided polygons, the average run time is less than 40 ms. Over 200 threats could be included within a second of computation.

   A trajectory tracker that explicitly accounts for turning rate and velocity constraints was integrated with a 12 degree-of-freedom (DOF) aircraft

model. Algorithm performance and behavior with this higher-order UAV model. The tracker was initially developed assuming a 6 DOF model.

The required bandwidth separation between layers in our hierarchy was carefully investigated. To address this, a simulation was developed in which the points passed to the trajectory smoother were changed randomly at every time step. Results show that no matter the input to the trajectory smoother, sensible outputs are generated. In fact, the output will always satisfy the heading rate and velocity constraints of the UAV. When the random point generation is switched to deliberate point planning, the smoother quickly tracks the path.

The algorithms were also tested via simulation using a cooperative multi-vehicle scenario where a team of unmanned air vehicles (UAVs) is given the task of searching a region with unknown opportunities and hazards. Opportunities and hazards were identified in-flight, within the immediate sensor radius of the UAVs, and are therefore constantly "popping-up." The team objective was to "visit" as many opportunities as possible, while "avoiding" as many hazards as possible. Since opportunities and hazards were constantly appearing on the sensor horizon, the waypoint paths that achieve the stated objective will also be changing in response to new information. It was assumed that the problem is essentially two dimensional so collision avoidance must be accounted for explicitly. A cooperative waypoint path planner for this scenario. The trajectory generator and trajectory tracker is currently being integrated with the path planner. This problem was used to study the bandwidth separation issue between the waypoint path planner and the trajectory generator.

2. An adaptive control law for the trajectory tracker was synthesized using Lyapunov based-nonlinear control techniques. The additional complexity added to the control law is minimal. The average run-time per sample period was 14.3 milliseconds for a 400 MHz Pentium II and 105 milliseconds for a 50 MHz Rabbit microcontroller. The initial algorithm was modified to explicitly account for velocity and heading rate constraints. A description of the algorithm will be submitted for possible presentation at the 2003 Conference on Decision and Control. The trajectory tracker was tested in simulation has also using mobile robots and a using full 12 degree-of-freedom (DOF) aircraft model.

3.4.2 Task 2: Algorithm Validation and Testing

A 12 DOF aircraft dynamic model was used to test the WPP and ATT algorithms via simulation. In porting the algorithms from Matlab/Simulink to C/C++ we remained cognizant of the fact that eventual deployment of the code will require extensive verification and validation. Therefore the WPP, DTS, and ATT code were designed with verification and validation in mind. To facilitate testing, all UAV and algorithm parameters were defined in configuration text files. This allowed a process to randomly generate different scenarios without

recompiling the code. It was in this way that Monte-Carlo simulations were performed. Algorithms performance issues analyzed include average speed, speed vs threat density, the existence of feasible path vs. threat density.

The results for the WPP algorithm on a 400MHz Pentium II (averaged over 200 random threat locations, for each threat density) are shown below in figure 2. In this figure, 100% density corresponds to 30 threats. It can be seen, that even at this level, the run-time is low enough that we can re-plan paths dynamically as threats pop-up and move dynamically.

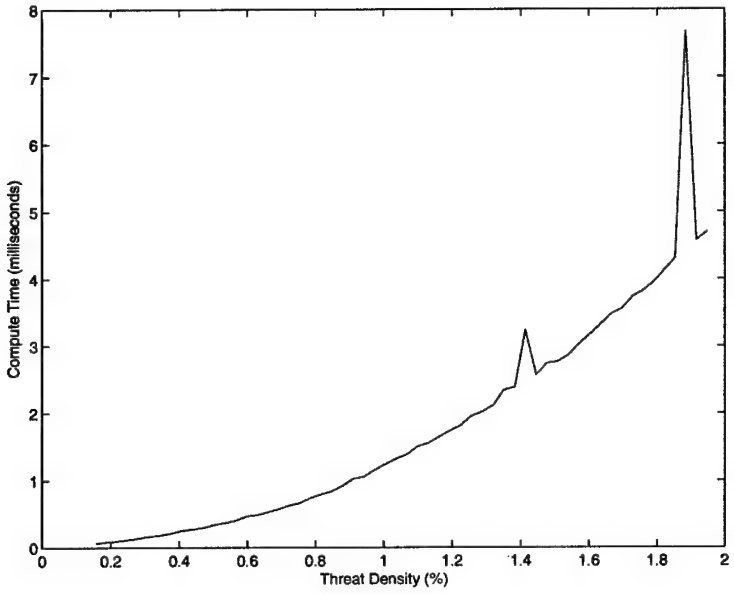


Figure 10: Compute time for Voronoi waypoint path planner verses threat density.

The upper bound of one iteration of the DTS algorithm on a 1.8 GHz Pentium 4 processor is 63 micro-seconds. This corresponds to approximately 3 milli-seconds on a 50 MHz processor. Run time for DTS is dependent only on the part of the turn that the UAV is executing. In most cases the time is negligible.

3.4.3 Task 3: Determine Implementation Requirements

As preparation for Phase II, system requirements for actual implementation in future combat were investigated. As part of this effort, the WPP and DTS Matlab/Simulink code were ported into real-time C code. The C/C++ implementation matches the Matlab implementation with functionality preserved.

The advantages of having the DTS implemented in C/C++ code are 2 fold: speed and size. On a Pentium 4 processor, one iteration of the algorithm has an upper bound of 63 micro-seconds. This roughly corresponds to 3 milliseconds on a 50 MHz Rabbit processor - the micro-processor onboard the prototype UAV. The code executable size is 82K, which easily fits the 1M of memory onboard the UAV. The DTS handles cases when the length of straight-line path segments is large compared to the turning radius of the UAV. Originally, the algorithm was sensitive to scale, but that problem has been solved and scale is determined by the size of the turning radius.

The VWPP has also been successfully ported to C/C++. The functionality and accuracy of the corresponding Matlab implementation has been matched. The speed of the VWPP is related to the number of threats. For 60 threats, the speed on a Pentium 4 is on average 5 milli-seconds.

As part of determining the real-time computational requirements of our approach, and fully understanding the implications of hardware implementation, in Phase I we worked to demonstrate the algorithms on a prototype UAV. An existing prototype computing platform at Brigham Young University has been adapted for real-time control system testing and evaluation of autonomous flight planning algorithms. (See Figure 11.) The physical properties of the aircraft

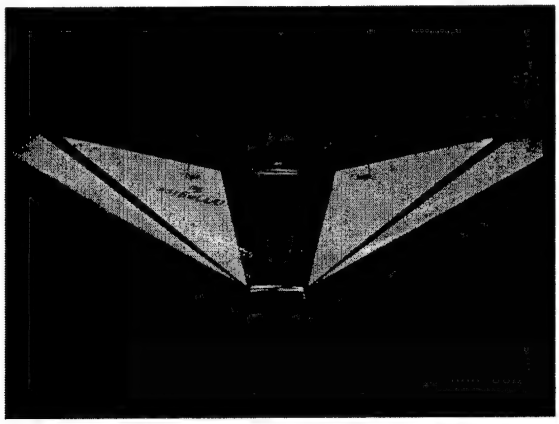


Figure 11: BYU Prototype UAV.

are shown in Table 3.4.3. The aircraft has a payload for 2lbs and an average airspeed of 35 mph. The UAV is equipped with a microcontroller and a full suite of sensors. In addition, a camera is mounted in the front of the camera. Communication to the UAV is accomplished through three separate RF channels. The first channel is a standard 70 MHz RF channel for remote control flight. The second channel is a 900 MHz RF modem with 2 mile range. This channel is used to communicate with the on-board computer and will be the primary channel for guidance, control, and telemetry. The third channel is a

Mass and Inertia			
I_x	0.1147	kgm^2	in $\mathcal{C}_b$
I_y	0.0576	kgm^2	in $\mathcal{C}_b$
I_z	0.1712	kgm^2	in $\mathcal{C}_b$
J_{xy}	0.0	kgm^2	in $\mathcal{C}_b$
J_{xz}	0.0015	kgm^2	in $\mathcal{C}_b$
J_{yz}	0.0	kgm^2	in $\mathcal{C}_b$
m	1.56	kg	
b	1.4224	m	wing span
S	0.4703	m^2	wing area
AR	4.3		aspect ratio

Table 1: Physical Parameters for the MAGICC lab UAV.

2.4 GHz channel that transmits the NTSC signal from the camera. The heart of the system is a Rabbit Core Module 3100 (RCM 3100) microcontroller that runs at 50 MHz and has 1 MB of code/data memory. The system has 22 digital I/O ports, 10 analog inputs, 4 serial TX/RX ports, 4 pulse width modulated outputs, 6 motor driving channels, and 4 quadrature encoder inputs. A bypass circuit can be used to switch between manual control and an on-board autopilot. The prototype UAV was used to test the trajectory generation algorithms is equipped with a full set of sensors which have been integrated on the UAV. A list of the sensors is shown in Table 3.4.3.

State variable	Sensor Type	Noise Variance	Sample Rate
x	GPS	1.0 (m)	1.0 s
y	GPS	1.0 (m)	1.0 s
h	absolute pressure	$x.x$ (m)	0.02 s
V	differential pressure	$x.x$ (m/s)	0.02 s
α	vane/encoder	$x.x$ (deg)	0.02 s
β	vane/encoder	$x.x$ (deg)	0.02 s
ϕ	Honeywell compass	0.5 (deg)	0.07 s
θ	Honeywell compass	0.5 (deg)	0.07 s
ψ	Honeywell compass	1.5 (deg)	0.07 s
p	rate gyro	$x.x$ (deg/s)	0.02 s
q	rate gyro	$x.x$ (deg/s)	0.02 s
r	rate gyro	$x.x$ (deg/s)	0.02 s

Table 2: Sensors used on the MAGICC lab UAV.

The initial direction was to employ an off-the-shelf autopilot for the aircraft. However, the particular product that was used (Micropilot), did not prove to be easy to interface with and a custom autopilot was developed instead. The autopilot has reached a level of maturity with successful demonstrations of altitude hold, heading hold, and velocity hold. The autopilot was integrated with PDA and voice interfaces and successfully demonstrated to representatives from

AFRL/MN and AFRL/VA. Video demonstrations are available at

http://www.ee.byu.edu/~ beard/public/uav/video/pda.avi
http://www.ee.byu.edu/~ beard/public/uav/video/voice.avi

3.5 Summary of Accomplishments in Phase I

A Voronoi-based path planner was developed and extended to account for polygon threat regions and no-fly zones. The robustness of trajectory smoother with respect to rapidly changing input was investigated and found to have no detrimental effect on performance. The computational complexity of the extended Voronoi path planner was analyzed and it was shown that for a reasonable number of threats, the planner will run in real-time for static and dynamic threat environments. A 12 DOF nonlinear model of the BYU UAV was developed in Simulink. The trajectory tracker and complete waypoint path planner were integrated and tested with the 12 DOF aircraft model.

In parallel with the software development, an autopilot was developed using the 12 DOF model and successfully flight-tested a on the BYU UAV. A light weight (2 ounce), low-cost, sensor board and flight computer for the was developed for the BYU UAV. The autopilot was deployed on the BYU UAV with with full implementation of heading hold, altitude hold, and velocity hold. The autopilot with also integrated with PDA and voice interfaces.

4 Plans for Phase II and III

In Phase I, we showed the feasibility of the WPP, DTS, and ATT algorithms for path planning and trajectory generation as well as develop the requirements for implementation in real-time environment for combat UAVs. Porting the algorithms to C in Task 2 lays the groundwork for Phase II, which will entail implementation, refinement, and demonstration of these algorithms.

Given a successful outcome of Phase I research and Phase II implementation, the technology proposed here can be readily commercialized leading to a trajectory planning software package suitable for acquisition by the Air Force. The algorithm codes will be optimized into a reliable, user-friendly software suite that will be compliant with the Tactical Control System to insure interoperablity with the family of all present and future tactical UAVs. Our efforts to develop this technology will be guided by cooperation with Lockheed-Martin and Northrop-Grumman.

Beyond military applications, the market for technology which enables machines to respond to change or the unexpected is developing. Agricultural and industrial demand for mobile autonomous robots is growing. Fuelled by increases in capability and decreases in price, industrial robot sales in the US have nearly tripled since 1991.[34] Path planning is an enabling technology for service robots which automate the collection of hazardous wastes in hospitals, as well

as, safe dismantlement of nuclear and chemical-biological weapons. Autonomous hotel vacuum cleaners are another potential area for commercialization of this technology.

ISL is pursuing two avenues of commercialization. The first is to partner with either Lockheed-Martin or Northrop-Grumman to develop autonomous route-planning software for future military UAV platforms such as LOCAAS.

We are also seeking backing of TeleDyne Controls to apply this technology to future commercial applications. ISL met with Teledyne Controls of Santa Monica, CA to determine their capabilities and requirements for implementation of autonomous navigation aids in flight control systems. A similar ongoing discussion is underway with the Honeywell Corp. Future flight management systems will record and share environmental information and weather conditions in real-time. On-board software could use this data to adjust propulsion and flight control systems to achieve optimized flight profiles. This would allow planes to change their routes in real-time to avoid turbulence and take advantage of favorable wind conditions while avoiding other aircraft.

5 Technology Transitions

BYU's initial trajectory generation research in cooperative control work led to Phase I of this STTR. The Phase I effort has directly resulted in several technology transfer opportunities. Recently, successful transition of technology developed in Phase I to activities of current and practical interest to the Air Force. has occurred. For example:

- **AFRL/MN.** Efforts to fly trajectories on UAV hardware under Phase I led to work with AFRL/MN to develop miniature autopilot hardware and software technologies that were recently used by Air Force Special Operations forces in war games in Mississippi.
- **NASA/Ames.** The waypoint path planning portion of our trajectory generation work has been utilized by the Army/NASA Rotorcraft Division in their Precision Autonomous Landing Adaptive Control Experiment (PALACE) program.
- **Spin-off Company.** The autopilot developed in part under Phase I has generated enough purchasing interest by university and government lab researchers that a small company has been created to manufacture and market autopilot hardware and software.

6 Publications

1. Erik P. Anderson, Randal W. Beard, "An Algorithmic Implementation of Constrained Extremal Control for UAVs," *AIAA Guidance and Control Conference*, Monterey, CA, August 2002, AIAA Paper no. 2002-4470.

2. Randal W. Beard, Timothy W. McLain, Michael Goodrich, Erik P. Anderson, "Coordinated Target Assignment and Intercept for Unmanned Air Vehicles," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 6, December, 2002, pp. 911–922.

3. Timothy W. McLain, Randal W. Beard, "Coordination Variables, Coordination Functions, and Cooperative Timing Missions," *American Control Conference,* Denver, CO, 2003, pp. 296–301.

4. Derek Kingston, Randal Beard, Timothy McLain, Michael Larsen, Wei Ren, "Autonomous Vehicle Technologies for Small Fixed Wing UAVs," *AIAA 2nd Unmanned Unlimited Systems, Technologies, and Operations–Aerospace, Land, and Sea Conference and Workshop & Exhibit,* San Diego, CA, September, 2003, Paper no. AIAA-2003-6559.

5. Randal W. Beard, Timothy W. McLain, "Multiple UAV Cooperative Search under Collision Avoidance and Limited Range Communication Constraints," *IEEE Conference on Decision and Control,* Maui, HA, 2003, (to appear).

6. Wei Ren, Randal W. Beard, "CLF-based Tracking Control for UAV Kinematic Models with Saturation Constraints," *IEEE Conference on Decision and Control,* Maui, HA, 2003, (to appear).

7. Erik P. Anderson, Randal W. Beard, Timothy W. McLain, "Real Time Dynamic Trajectory Smoothing for Uninhabited Aerial Vehicles," *IEEE Transactions on Control Systems Technology*, to appear.

8. Wei Ren, Randal W. Beard, "Trajectory Tracking for Unmanned Air Vehicles with Velocity and Heading Rate Constraints," *IEEE Transactions on Control Systems Technology*, to appear.

9. Wei Ren, Randal W. Beard, "Constrained Nonlinear Tracking Control For Small Fixed-wing Unmanned Air Vehicles," *2004 American Control Conference*, in review.

10. Tim McLain, Randal W. Beard, "Unmanned Air Vehicle Testbed for Cooperative Control Experiments," *2004 American Control Conference*, (invited) in review.

11. Derek B. Kingston, Randal W. Beard, "Real-Time Attitude and Position Estimation for Small UAVs," *2004 American Control Conference*, in review.

7 Personnel Supported

Michael Larsen, Corporate PI, ISL, `mlarsen@islinc.com`
Randal Beard, University PI, BYU, `beard@ee.byu.edu`

Timothy McLain, University CO-PI, BYU, tmclain@et.byu.edu
Joshua Hintz, Research Assistant, BYU, hintzej@ee.byu.edu
Derek Kingston, Research Assistant, BYU, derek_kingston@hotmail.com
Walter Johnson, Research Assistant, BYU, whj@ee.byu.edu
David Hubbard, Research Assistant, BYU, dch24@email.byu.edu

8 References

[1] R. W. Beard, T. W. McLain, M. Goodrich, and E. P. Anderson, "Coordinated target assignment and intercept for unmanned air vehicles," *IEEE Transactions on Robotics and Automation*, (submitted November 2001).

[2] *Uninhabited Air Vehicles: Enabling Science for Military Systems.* National Academy Press, 2000.

[3] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Proababilistic roadmaps for path planning in high dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[4] R. Bohlin and L. Kavraki, "Path planning using lazy rpm," in *Proceedings of the IEEE International Conference on Robotics an Automation*, 1997.

[5] S. LaValle and J. Kuffner, "Randomized kinodynamic planning," in *Proceedings of the 1999 IEEE International Conference on Robotics and Automation*, 1999.

[6] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Technical Report 98-11, Iowa State University, Ames,IA, Oct. 1998.

[7] E. Frazzoli, M. Dahleh, and E. Ferron, "Real-time motion planning for agile autonomous vehicles," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, (Denver, CO), August 2000. AIAA Paper No. AIAA-2000-4056.

[8] E. Frazzoli, M. Dahleh, and E. Ferron, "Robust hybrid control for autonomous vehicle motion planning," Technical Report LIDS-P-2468, Massachusetts Institute of Technology, Cambridge, MA, Oct. 1999. submitted to the *IEEE Transactions on Automatic Control.*

[9] C. Hargraves and S. Paris, "Direct trajectory optimization using nonlinear programming and collocation," *AIAA J. Guidance and Control*, vol. 10, pp. 338–342, 1987.

[10] O. von Stryk and R. Bulirsch, "Direct and indirect methods for trajectory optimization," *Annals of Operation Research*, vol. 37, pp. 357–373, 1992.

[11] Y. Chen and J. Huang, "A new computational approach to solving a class of optimal control problems," *International Journal of Control*, vol. 58, no. 6, pp. 1361–1383, 1993.

[12] L. Singh and J. Fuller, "Trajectory generation for a UAV in urban terrain, using nonlinear MPC," in *Proceedings of the American Control Conference*, (Arlington, VA), 2001.

[13] A. Isidori, *Nonlinear Control Systems*. Communication and Control Engineering, New York, New York: Springer Verlag, 2nd ed., 1989.

[14] B. Charlet, J. Levine, and R. Marino, "On dynamic feedback linearization," *Systems and Control Letters*, vol. 13, pp. 143–151, 1989.

[15] M. Fliess, J. Levine, P. Martin, and P. Rouchon, "Flatness and defect of non-linear systems: introductory theory and examples," *International Journal of Control*, vol. 61, no. 6, pp. 1327–1360, 1995.

[16] M. Milam and R. M. K. Mushambi, "A new computational approach to real-time trajectory generation for constrained mechanical systems," in *Proceedings of the IEEE Conf. on Decision and Control*, (Sydney, Australia), pp. 845–851, December 2000.

[17] M. Milam, R. Franz, and R. Murray, "Real-time constrained trajectory generation applied to a flight control experiment," in *Proceedings of the International Federation of Automatic Control Conference*, 2002.

[18] S. Agrawal and N. Faiz, "Optimization of a class of nonlineaer dynamic systems: new efficient method without Lagrange multipliers," *J. Optimization Theory and Applications*, vol. 97, no. 1, pp. 11–28, 1998.

[19] N. Faiz, S. K. Agrawal, and R. M. Murray, "Trajectory planning of differentially flat systems with dynamics and inequalities," *AIAA Journal of Guidance, Control and Dynamics*, vol. 24, pp. 219–227, March–April 2001.

[20] G. J. Toussaint, T. Basar, and F. Bullo, "Motion planning for nonlinear underactuated vehicles using H^∞ techniques," in *American Control Conference*, 2001. In review.

[21] R. Sedgewick, *Algorithms*. Addison-Wesley, 2nd ed., 1988.

[22] D. Eppstein, "Finding the k shortest paths," *SIAM Journal of Computing*, vol. 28, no. 2, pp. 652–673, 1999.

[23] T. McLain and R. Beard, "Cooperative rendezvous of multiple unmanned air vehicles," in *Proceedings of the AIAA Guidance, Navigation and Control Conference*, (Denver, CO), August 2000. Paper no. AIAA-2000-4369.

[24] P. Chandler, S. Rasumussen, and M. Pachter, "UAV cooperative path planning," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, (Denver, CO), August 2000. AIAA Paper No. AIAA-2000-4370.

[25] R. W. Beard, T. W. McLain, M. Goodrich, and E. P. Anderson, "Coordinated target assignment and intercept for unmanned air vehicles," *IEEE Transactions on Robotics and Automation*, vol. 18, pp. 911–922, December 2002.

[26] W. Ren and R. W. Beard, "CLF-based tracking control for UAV kinematic models with saturation constraints," in *Proceedings of the IEEE Conference on Decision and Control*, 2003. (to appear).

[27] W. Ren and R. W. Beard, "Trajectory tracking for unmanned air vehicles with velocity and heading rate constraints," *IEEE Transactions on Control Systems Technology*, (in review).

[28] T.-C. Lee, K.-T. Song, C.-H. Lee, and C.-C. Teng, "Tracking control of unicycle-modeled mobile robots using a saturation feedback controller," *IEEE Transactions on Robotics and Automation*, vol. 9, pp. 305–318, March 2001.

[29] E. P. Anderson and R. W. Beard, "An algorithmic implementation of constrained extremal control for UAVs," in *Proceedings of the AIAA Guidance, Navigation and Control Conference*, (Monterey, CA), August 2002. AIAA Paper No. 2002-4470.

[30] E. P. Anderson, R. W. Beard, and T. W. McLain, "Real time dynamic trajectory smoothing for uninhabited aerial vehicles," *IEEE Transactions on Control Systems Technology*, (in review).

[31] E. P. Anderson, "Constrained extremal trajectories and unmanned air vehicle trajectory generation," Master's thesis, Brigham Young University, Provo, Utah 84602, April 2002.

[32] R. L. Burden and J. D. Faires, *Numerical Analysis*. Boston: PWS-KENT Publishing Company, fourth edition ed., 1988.

[33] T. Balch and R. C. Arkin, "Behavior-based formation control for multirobot teams," *IEEE Transactions on Robotics and Automation*, vol. 14, pp. 926–939, December 1998.

[34] "The big comeback in robots," *Fortune Magazine*, pp. 11–28, March 4 1996.

[35] R. W. Beard, T. W. McLain, M. Goodrich, and E. P. Anderson, "Coordinated target assignment and intercept for unmanned air vehicles," *IEEE Transactions on Robotics and Automation*, vol. 18, pp. 911–922, December 2002.

[36] T. W. McLain and R. W. Beard, "Coordination variables, coordination functions, and cooperative timing missions," in *Proceedings of the American Control Conference*, 2003. to appear.

[37] T. W. McLain, P. R. Chandler, S. Rasmussen, and M. Pachter, "Cooperative control of UAV rendezvous," in *Proceedings of the American Control Conference*, (Arlington, VA), pp. 2309–2314, June 2001.

[38] T. McLain, P. Chandler, S. Rasmussen, and M. Pachter, "Cooperative control of UAV rendezvous," in *Proc. of the ACC*, June 2001.

[39] T. W. McLain, R. W. Beard, and J. Kelsey, "Experimental demonstrations of multiple robot cooperative target intercept," in *Proceedings of the AIAA Guidance, Navigation and Control Conference*, (Monterey, CA), August 2002. AIAA Paper No. 2002-4678.

9 Appendix 1: Application to Coordinated Control

Future research on UAVs will increasingly focus on cooperative control scenarios involving multiple vehicles. Examples include cooperative timing, search, and ISR missions, and formation flight. Solutions to cooperative control problems will increasingly rely on real-time path planning and trajectory generation techniques. For example, in cooperative timing problems it is necessary to search through a set of paths that are jointly feasible for the UAV team. Therefore, computationally efficient techniques for path planning become essential. As another example, cooperative search and ISR scenarios are inherently coupled at the path/trajectory level. In addition, multiple vehicles in a congested battlefield necessitate collision avoidance and deconfliction techniques at the path/trajectory level.

Therefore, computationally efficient real-time path planning and trajectory generation techniques directly impact the state-of-the-art in cooperative control. In fact, the path planning and trajectory generation techniques funded under Phase I were originally developed to facilitate solutions to cooperative timing problems. In this regards, a key feature of the path planning/trajectory generation techniques is that the real-time trajectories have the same path length as the waypoint paths, and thus satisfy the same timing constraints. The result is that cooperative timing missions can be planned using waypoint paths, significantly reducing the computational overhead.

Applications of the techniques developed in Phase I to cooperative control scenarios is reported in.[23,29–31,35–39]

10 Appendix 2: Algorithm Codes for 12 Degree of Freedom Simulation

The algorithm codes for the Matlab/C simulation of the 12 DOF UAV model of the way point path planner, trajectory smoother, and tracker are provided on a CD accompanying the report. A hard copy of the files is included as well.

10.1 Matlab Script Files

```
function y = autopilot(u,Ts,auto)
% autopilot
%
% autopilot function for MAGICC lab UAV
%
% internal function states will be labeled X.name
%
% Modification History:
%   3/28/03 - RWB
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% inputs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
psi_c = u(1);                % heading command
%phi_c = u(1);                % roll command
V_c   = u(2);                % velocity command
h_c   = u(3);                % altitude command
x     = u(4);    % inertial x-position
y     = u(5);    % inertial y-position
h     = u(6);    % altitude
V     = u(7);    % velocity (air speed)
alpha = u(8);    % angle of attack
beta  = u(9);    % sideslip angle
phi   = u(10);   % roll angle
theta = u(11);   % pitch angle
psi   = u(12);   % heading angle
p     = u(13);   % roll rate
q     = u(14);   % pitch rate
r     = u(15);   % yaw rate
t     = u(16);   % time

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% implement autopilot modes

% velocity autopilot
```

```
delta_t = velocity_hold(V, V_c, Ts, t);

% longitudinal autopilots
if 1,
  theta_c = altitude_hold(h, h_c, Ts, t);
  q_c     = 0;
  u       = V*cos(alpha)*cos(beta);
  w       = V*sin(alpha)*cos(beta);
  x       = [u; w; q; theta];
  xd      = [V_c; 0; q_c; theta_c];
  delta_e = longitudinal_autopilot(x, xd, t, Ts, auto);
else
  theta_c  = altitude_hold(h, h_c, Ts, t);
  q_c     = pitch_attitude_hold(theta, theta_c, Ts, t);
  delta_e = pitch_rate_hold(q, q_c, Ts, t);
end

% lateral autopilots
if 1,
  phi_c   = heading_hold(psi, psi_c, Ts, t);
  p_c      = 0;
  x       = [beta; p; r; phi];
  xd      = [0; 0; 0; phi_c];
  [delta_a, delta_r] = lateral_autopilot(x, xd, t, Ts, auto);
else
  phi_c   = heading_hold(psi, psi_c, Ts, t);
  p_c     = roll_attitude_hold(phi, phi_c, Ts, t);
  delta_a = roll_rate_hold(p, p_c, Ts, t);
  delta_r = side_slip_hold(beta, r, Ts, t);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mix aileron and elevator commands
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% control outputs
u = [...
        delta_e + delta_a;... % delta_er: right elevon
        delta_e - delta_a;... % delta_el: left elevon
        delta_r;...           % delta_rr: right rudder
        delta_r;...           % delta_rl: left rudder
        delta_t;...           % delta_t: thrust
    ];

% desired states (for visulation and debugging)
xd = [...
        0;...       % x: desired x-position (not accounted for y autopilot)
```

```
        0;...       % y: desired y-position (not accounted for y autopilot)
        h_c;...     % h: desired altitude
        V_c;...     % V: desired velocity
        0;...       % alpha: desired angle of attack
        0;...       % beta:desired side slip angle
        phi_c;...   % phi: desired roll angle
        theta_c;... % theta: desired pitch angle
        psi_c;...   % psi:  desired yaw angle
        %0; ...
        p_c;...     % desired roll rate
        q_c;...     % desired pitch rate
        0;...     % desired yaw rate
    ];

% output
y = [u; xd];



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Autopilot functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% lateral autopilot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [delta_a, delta_r] = lateral_autopilot(x, xd, t, Ts,
auto);
  if isempty(auto.L_lat), % no observer
    u = -auto.K_lat*(x-xd);
  else
    % initialize the observer state
    persistent X;
    if isempty(X)|(t<2*Ts), X.xhat = zeros(4,1); end
    X.xhat = auto.A_lat_d*X.xhat + auto.B_lat_d*(x-xd);
    u = -auto.K_lat*(X.xhat);
  end

  delta_a = u(1);
  delta_r = u(2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% longitudinal autopilot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function delta_e = longitudinal_autopilot(x, xd, t, Ts, auto);
  if isempty(auto.L_lon), % no observer
    delta_e = -auto.K_lon*(x-xd);
  else
    % initialize the observer state to initial xd
    persistent X;
    if isempty(X)|(t<2*Ts), X.xhat = zeros(4,1); end
    X.xhat = auto.A_lon_d*X.xhat + auto.B_lon_d*(x-xd);
    delta_e = -auto.K_lon*(X.xhat);
  end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% altitude_hold
%  - regulate altitude
%  - produces desired pitch angle
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function theta_c = altitude_hold(h, h_c, Ts, t);
  persistent X;
  if isempty(X)|(t<2*Ts), X.integrator = 0; end

  % error equation
  e_h = h_c - h;

  A = 2;
  B = 30;

  if e_h > A,
      theta_c = B*pi/180;
      X.integrator = 0;
  elseif e_h < -A
      theta_c = -B*pi/180;
      X.integrator = 0;
  else
    if abs(e_h) > 0.5*A,
       X.integrator = 0;
    else
      % update integrator
      X.integrator = X.integrator + Ts*e_h;
    end
    % implement PI control
    kp      = B*pi/180/A;
    ki      = .01;
    theta_c = (kp * e_h + ki * X.integrator);
  end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% pitch_attitude_hold
%  - regulate pitch_attitude
%  - produces desired pitch rate
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function q_c = pitch_attitude_hold(theta, theta_c, Ts, t);
  persistent X;
  if isempty(X)|(t<2*Ts),
      X.integrator = 0;
      X.theta_prev = 0;
      X.theta_filtered_prev = 0;
  end

  % implement digital washout filter
  Kw = .01;           % washout filter gain
%  Kw = 0;           % washout filter gain
  kw1 = 1/(1+(Ts*Kw/2));
  kw2 = (1-(Ts*Kw/2))*kw1;
  theta_filtered = kw2 * X.theta_filtered_prev...
       + kw1 * (theta - X.theta_prev);

  % error equation
  e_theta = theta_c - theta_filtered;

  % update persistent variables
  X.integrator = X.integrator + Ts*e_theta;
  X.theta_filtered_prev = theta_filtered;
  X.theta_prev = theta;

  % implement PI control
  kp      = 10;
  ki      = 5;
  q_c = kp * e_theta + ki * X.integrator;

  % saturate pitch rate
  q_c = sat(q_c, 100*pi/180, -100*pi/180);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% pitch_rate_hold
%  - regulate pitch_rate
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function delta_e = pitch_rate_hold(q, q_c, Ts, t);
  persistent X;
  if isempty(X)|(t<2*Ts), X.integrator = 0; end

  % error equation
```

```
  e_q = q_c - q;

  % update integrator
  X.integrator = X.integrator + Ts*e_q;

  % implement P control
  kp      = 0.65;
  ki      = 1;
  delta_e = kp * e_q + ki * X.integrator;

  % saturate the output
  delta_e = -sat(delta_e, 45*pi/180, -45*pi/180 );



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% heading hold
%  - regulate heading angle
%  - produces desired roll angle
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function phi_c = heading_hold(psi, psi_c, Ts, t);
  persistent X;
  if isempty(X)|(t<2*Ts), X.integrator = 0; end

  % error equation
  e_psi = psi_c - psi;

  A = 10;
  B = 20;

  if e_psi > A*pi/180,
      phi_c = B*pi/180;
      X.integrator = 0;
  elseif e_psi < -A*pi/180,
      phi_c = -B*pi/180;
      X.integrator = 0;
  else
    if abs(e_psi) > A/5*pi/180,
        X.integrator = 0;
    else
      % update integrator
      X.integrator = X.integrator + Ts*e_psi;
    end
    % implement PI control
    kp      = B/A;
    ki      = 0;
```

```
    phi_c = kp * e_psi + ki * X.integrator;
  end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% roll_attitude_hold
%  - regulate roll_attitude
%  - produces desired roll rate
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function p_c = roll_attitude_hold(phi, phi_c, Ts, t);
  persistent X;
  if isempty(X)|(t<2*Ts), X.integrator = 0; end

  % error equation
  e_phi = phi_c - phi;

  A = 3;
  B = 10;

  if e_phi > A*pi/180,
      p_c = B*pi/180;
      X.integrator = 0;
  elseif e_phi < -A*pi/180,
      p_c = -B*pi/180;
      X.integrator = 0;
  else
    if abs(e_phi) > A/5*pi/180,
        X.integrator = 0;
    else
      % update integrator
      X.integrator = X.integrator + Ts*e_phi;
    end
    % implement PI control
    kp      = B/A;
    ki      = 0;
    p_c = kp * e_phi + ki * X.integrator;
  end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% roll_rate_hold
%  - regulate roll_rate
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function delta_a = roll_rate_hold(p, p_c, Ts, t);
  persistent X;
  if isempty(X)|(t<2*Ts), X.integrator = 0; end
```

```
  % error equation
  e_p = p_c - p;

  % update integrator
  X.integrator = X.integrator + Ts*e_p;

  % implement P control
  kp      = 5;
  ki      = 2;
  delta_a = kp * e_p + ki * X.integrator;

  delta_i = p_c - kp*p;

  % saturate the output
  delta_a = -sat(delta_a, 30*pi/180, -30*pi/180 );


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% side_slip_hold
%  - hold the side slip angle to zero
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function delta_r = side_slip_hold(beta, r, Ts, t);
  persistent X;
  if isempty(X)|(t<2*Ts), X.integrator = 0; end

  % update integrator
  X.integrator = X.integrator + Ts*beta;

  % implement PI control
  kp      = 20;
  ki      = 1;
  kd      = -1;
  delta_r = kp * beta + ki * X.integrator + kd*r;

  % saturate the output
  delta_r = -sat(delta_r, 10*pi/180, -10*pi/180 );

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% velocity_hold - PI control to hold velocity using throttle only
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function delta_t = velocity_hold(V, V_c, Ts, t);
  persistent X;

  % convert inputs to general variables
  y = V;
  r = V_c;
```

```
% controller constants
am   = .5; % approximately 1.5 second rise time
bm   = am;
gam1 = 1;
gam2 = 1;

if isempty(X)|(t<2*Ts),
    X.ym = r;
    X.th1 = bm/am;
    X.th2 = 1/am;
end

% output control
u = X.th1*y + X.th2*(am*r - bm*y + gam1*(X.ym-y));

% update state equations
X.ym  = X.ym + Ts*( -bm*X.ym + am*r );
X.th1 = X.th1 + Ts*( gam1*y*(X.ym-y) );
X.th2 = X.th2 + Ts*( (X.ym-y)*(am*r - bm*y + gam1*(X.ym-y)) );


% saturate the output
delta_t = sat(u, 10, -10 );

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% sat
%   - saturation function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function out = sat(in, up_limit, low_limit);
  if in > up_limit,
      out = up_limit;
  elseif in < low_limit;
      out = low_limit;
  else
      out = in;
  end
  function ic = calcInitialCond;
  global P;
  global uavdata;
  global env;
  global M;

  for p=1:P,
     uav(p).path = []; % 3 x n [path [x; y; h]]
     uav(p).tg = 0;
```

```
   uav(p).st = 0;
   startpts = [reshape(uavdata(p).ic(1:2),1,2); env.targets(:,:,p)]; % each row
   while size(uav(p).path,2) < 3,
      if uav(p).tg+1 > size(env.targets(:,:,p),1),
         uav(p).tg = 1;
      else,
         uav(p).tg = uav(p).tg + 1;
      end;
      if uav(p).st+1 > size(startpts,1),
         uav(p).st = 2;
      else,
         uav(p).st = uav(p).st + 1;
      end;

      vel = (uavdata(p).v_max + uavdata(p).v_min)/2; % pick velocity inbetween min and ma
      [vp, vel] = generateVPpath(env.threats, startpts(uav(p).st,:), env.targets(uav(p).t
      pathlen = size(vp,2);
      uav(p).path = [ uav(p).path [vp(2:3,:); uavdata(p).ic(5)*ones(1,pathlen)] ];
      uav(p).vel = vel;
   end;
end;

%%%%%%%%%%%%%%%%%%%% target manager %%%%%%%%%%%%%%%%%%%%%%%%%
% initial targets
targets = env.targets(uav(1).tg,:,1)';
for p=2:P,
  targets = [targets; env.targets(uav(p).tg,:,p)'];
end

% initialize target manager state
ic.targetmgr_init = [...
      ones(P,1);...          % initial state of state machine
      targets;...            % initial target locations
      ones(P,1);...          % initial target pointers
  ];
ic.targetmgr_in = zeros(P,1);

%%%%%%%%%%%%%%%%%%%%     wpp        %%%%%%%%%%%%%%%%%%%%%%%%%
ic.wpp_init = [...
      size(env.threats,1);...  % number of threats
      ones(P,1);...            % initial state of state machine
  ];
for p=1:P,
      [m, n] = size(uav(p).path);
      if m*n > 3*M,
          disp('Must increase M to at least: ');
```

```
        m*n
    end;
    uav(p).padded_path = [reshape(uav(p).path,m*n,1); zeros(3*M - m*n,1)];
    uav(p).pathlen = n;
    ic.wpp_init = [ic.wpp_init; ...
            n; ...                      % number of wps in path
            uav(p).vel; ...             % velocity along path
            uav(p).padded_path; ...     % path with zeros
        ];
end;
ic.wpp_init = [ic.wpp_init; ...
                targets; ...    % initial targets
                zeros(P,1);...  % initial flag value
  ];

%%%%%%%%%%%%%%%%%%%%  waypoint mng   %%%%%%%%%%%%%%%%%%%%%%%%%%
for p=1:P,
    ic.wpmng_init(p).x0 = [ ...
      1;...                         % state;...
      0;...                         % request_new_path;...
      uavdata(p).ic(1:3);...        % w0;...
      1;...                         % wpPtr;...
      uav(p).padded_path; ...
      zeros(3*M,1); ...             % wpQueue;...
      size(env.threats,1);...       % num_threats_prev;...
      0;...                         % counter
      uav(p).pathlen;...            % path_len
      uav(p).padded_path;...        % stored_path
      uav(p).pathlen;...            % stored_path_len
    ];
end;

for p=1:P,
    ic.wpmng_in(p).in0 = [ ...
            uav(p).pathlen; ...
            uav(p).vel; ...
            uav(p).padded_path; ...
        ];
end;

%%%%%%%%%%%%%%%%%%%%  traj Gen     %%%%%%%%%%%%%%%%%%%%%%%%%%
for p=1:p,
    ic.trajGen_in(p).in0 = [uav(p).vel; uav(p).padded_path(1:9)];
    uavdata(p).ic(4) = uav(p).vel;
end;
function [sys,x0,str,ts] = constrained_tt(t,x,u,flag,uavdata,P)
```

```
% s-function implementation of adaptive trajectory tracker
%    without adaptation.
%
% Modified: 9/26/02 - RWB
%

switch flag,

  %%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes(P);

  %%%%%%%%%%%
  % Outputs %
  %%%%%%%%%%%
  case 3,
    sys=mdlOutputs(t,x,u,uavdata,P);

  %%%%%%%%%%%%%%%%%%%
  % Unhandled flags %
  %%%%%%%%%%%%%%%%%%%
  case { 1, 2, 4, 9 },
    sys = [];

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end
% end csfunc

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes(P)

sizes = simsizes; sizes.NumContStates  = 0; sizes.NumDiscStates  =
0; sizes.NumOutputs     = 3*P; sizes.NumInputs      = 11*P;
sizes.DirFeedthrough = 1; sizes.NumSampleTimes = 1;
```

```
sys = simsizes(sizes); x0 = []; str = []; ts  = [0 0];

% end mdlInitializeSizes

%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function yy=mdlOutputs(t,xx,u,uavdata,P, tparam) yy = [];
% interpret states and inputs
for p=1:P,
    eps_vmin = uavdata(p).track_eps_vmin;
    eps_vmax = uavdata(p).track_eps_vmax;
    eps_wmax = uavdata(p).track_eps_wmax;
    vmin  = uavdata(p).v_min - eps_vmin;
    vmax  = uavdata(p).v_max + eps_vmax;
    wmax  = uavdata(p).psidot_max + eps_wmax;

    x_r      = u(1+11*(p-1));
    y_r      = u(2+11*(p-1));
    psi_r    = u(3+11*(p-1));
    V_r      = sat(u(4+11*(p-1)),vmax-eps_vmax,vmin+eps_vmin);
    w_r      = sat(u(5+11*(p-1)),wmax-eps_wmax,-wmax+eps_wmax);
    x        = u(6+11*(p-1));
    y        = u(7+11*(p-1));
    psi      = u(8+11*(p-1));
    V        = u(9+11*(p-1));
    h        = u(10+11*(p-1));
    hdot     = u(11+11*(p-1));

    cp = cos(psi);
    sp = sin(psi);

    x0 =  angle_diff( psi_r, psi );
    x1 = -sp*( x_r - x ) + cp*( y_r - y );
    x2 = -cp*( x_r - x ) - sp*( y_r - y );
    cx0 = cos(x0);

    %x0bar = 100.0*x0+x1/sqrt(1+x1*x1+x2*x2);
    x0bar = x0+x1/sqrt(1+x1*x1+x2*x2);

    u1_up  = vmax - V_r*cx0;
    u1_low = vmin - V_r*cx0;
```

```
    u0 = -sat(10*x0bar,eps_wmax,-eps_wmax);
    u1 = sat(-10*x2,u1_up,u1_low);

    % compute actual controls
    %w = w_r;
    %v = V_r;
    w = w_r - u0;
    v = V_r*cx0 + u1;
    psi_c = psi + w*uavdata(p).tau_psi;

    % command roll angle
    %psi_c = atan2(w*v, 9.80665);

    yy = [yy; psi_c; v; w];
end;

function nval = sat(val, mx, mn)
  if val > mx, nval = mx;
  elseif val < mn, nval = mn;
  else, nval = val; end;


function err = angle_diff(aref, areal)
  err = aref - areal;
  while err >  pi, err = err - 2*pi; end;
  while err < -pi, err = err + 2*pi; end;%% dof12param.m
%%
%% Parameter file for MAGICC lab UAV
%%
%% Modified:
%%   10/31/02 - RWB
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for p=1:P
   % initial state
   dof12simulator(p).x0 = [...
        uavdata(p).ic(1);...          % X:     x-position (m)
        uavdata(p).ic(2);...          % Y:     y-position (m)
        uavdata(p).ic(5);...          % h:     altitude (m)
        uavdata(p).ic(4);...          % V:     velocity (m/s)
        2.4*pi/180;...                % alpha: angle of attack (rad)
        0;...                         % beta:  sideslip angle (rad)
        0;...                         % phi:   roll angle
        2.4*pi/180;...                % theta: pitch angle
        uavdata(p).ic(3);...          % psi:   yaw angle
```

```
        0;...                          % p:     roll rate
        0;...                          % q:     pitch rate
        0;...                          % r:     yaw rate
    ];
end;


inches2meters = 2.54/100;
% items we need to know to calculate dynamic pressure
C.ps_inches        = 30.02;                              % air pressure in inches-Hg
C.ps               = C.ps_inches*100*1.01e5/39.4/76;  % (N/m^2) air pressure
C.T_F              = 41;                                 % Temperature in Fharenheit
C.T                = 5/9*(C.T_F-32)+273.15;              % Temperature in Kelvin

% Physical parameters of aircraft
C.wingspan         = 1.4224;  % (meters) = 56 inches *2.54/100
C.root_chord       = 0.4572;  % (meters) = 18 inches *2.54/100
C.tip_chord        = 0.2032;  % (meters) = 8 inches *2.54/100
C.angle_of_sweep   = 0.6632;  % (rad) = 38 degrees *pi/180
C.m                = 1.56;    % (kg) mass = 55 ounces *28.35/1000
C.wing_loading     = 3.325;   % kg/m^2 = 10.9 ounces/ft^2 *28.35/1000*3.28^2
C.wing_area        = 0.4703;  % (m^2) = 5.06 ft^2 /3.28/3.28
C.cg               = 0.3089;  % (m) center of gravity behind LE at root
C.required_washout = 0.01506; % (m) Tip TE dist rise from LE
C.stability_factor = 0.043;   % ??
C.K1               = 0.622;   % ??
C.K2               = 0.378;   % ??
C.J = [...                    % inertia matrix in kg-m^2  (rough estimate)
      12.2, 0, 0.16;...
      0, 6.12, 0;...
      0.16, 0, 18.2;...
    ]*(14.6/(39.4^2));



% Airfoil specs
C.airfoil_type         = 'EH2010/EH2012';
C.root_zero_lift_angle = 0.0742*pi/180; % (radians) = 0.0742 degrees *pi/180
C.root_pitch_moment    = 0.0001;
C.tip_zero_lift_angle  = 0.0743*pi/180; % (radians) = 0.0743 degrees *pi/180
C.design_lift_coef     = 0.4255; C.aspect_ratio         = 4.3;
C.mean_chord           = 0.3302;        % (meters) = 13 inches *2.54/100
C.mean_chord_winglet   = 6*inches2meters; C.taper_ratio          =
0.444;
C.alpha_total          = -0.0733;       % (radians) = -4.2 degrees *pi/180
C.alpha_modified       = -0.0733;       % (radians) = -4.2 degrees *pi/180
```

```
C.aerodynamic_center   = 0.3226;        % (meters) = 12.7 inches behind LE at root
C.S = C.wing_area;
C.S_elevon             = 0.03;          % (m^2) = 46.5 in^2
C.elevon_mean_chord    = 0.0549;        % (m)   = 2.16 in

% flying properties
C.v_ave          = 11.18; % (m/s) = 25 miles/hr*1.16*1000/60/60

% physical parameters
C.g      = 9.80665;     % (m/s^2) gravitational constant
C.lambda = -0.0065;     % (Kelvin/m) temperature gradient in troposphere
C.Ra     = 8314.32;     % (J/K/kmol) molar gas constant
C.p0     = 101325;      % (N/m) air pressure at sea level
C.R      = 287.05;      %(J/K/kg) specific gas constant
C.M0     = 28.9644;     % (kg/kmole) molecular weight of air at sea level
C.Rearth = 6371020;     % (m) radius of earth
C.rho    = C.ps/C.R/C.T;     % (kg/m^3) air density

% aerodynamic coefficients
  C.b = C.wingspan;
  C.cbar = C.mean_chord;

  % coefficients from the airframe
  C.bw    = 56*inches2meters;
  C.aw    = C.bw/2*sin(38*pi/180);
  C.cw    = 10.66*inches2meters;
  C.dw    = 18*inches2meters;
  xw      = max(roots([(C.cw-C.dw)/(C.bw/2), -C.dw*C.bw/2, -(C.cw-C.dw)*C.bw/4]));
  C.bwl   = 12*inches2meters;
  C.cwl   = 4*inches2meters;
  C.dwl   = 8*inches2meters;
  xwl     = max(roots([(C.cwl-C.dwl)/(C.bwl/2), -C.dwl*C.bwl/2, -(C.cwl-C.dwl)*C.bwl/4]));
  C.R_cg  = [-12.16*inches2meters; 0; 0];  % center of gravity wrt leading edge
  acw      = 12.7*inches2meters;
  acwl     = xwl*sin(38*pi/180) + (C.dwl + (C.cwl-C.dwl)/(C.bwl/2)*xwl)/4;
  C.R_ac  = [-acw; 0; 0];                  % aerodynamic center wrt leading edge
  C.R_rw  = C.R_ac + [0; xw; 0];
  C.R_lw  = C.R_ac + [0; -xw; 0];
  C.R_rwl = [-C.aw-acwl; C.bw/2; -xwl];
  C.R_lwl = [-C.aw-acwl; -C.bw/2; -xwl];

  % aerodynamics coefficient for force model (From ModelFoil Version 4.1)
    % wing coefficients
    C.C_Lw_0     = 0;
    C.C_Lw_alpha = 0.1032*(180/pi);
    C.C_Lw_delta = 0.0509*(180/pi);
```

```
  C.C_Dw_0     = 0.0130;
  C.C_Dw_K     = -0.0032;
  C.C_Mw_0     = 0;
  C.C_Mw_alpha = -0.0016*(180/pi);
  C.C_Mw_delta = -0.0117*(180/pi);
  C.C_l_p      = -1;  % moment due to roll rate
  C.C_m_q      = -.5;  % moment due to pitch rate

  %winglet coefficients
  C.C_Lwl_0     = 0;
  C.C_Lwl_alpha = 0.1032*(180/pi);
  C.C_Lwl_delta = 0.0509*(180/pi);
  C.C_Dwl_0     = 0.0130;
  C.C_Dwl_K     = -0.0032;
  C.C_Mwl_0     = 0;
  C.C_Mwl_alpha = -0.0016*(180/pi);
  C.C_Mwl_delta = 0.0117*(180/pi);

  % engine coefficient
  C.C_engine = 1;

% aerodynamic coefficents for linear force model
% longitudinal coefficents
C.C_X_0        = -0.03554;
C.C_X_alpha    = 0.002920;
C.C_X_q        = -0.6748;
C.C_X_elevator = 0.03412;
C.C_X_throttle = 1;
C.C_Z_0        = -0.05504;
C.C_Z_alpha    = -5.578;
C.C_Z_q        = -2.988;
C.C_Z_elevator = -0.3980;
C.C_Z_throttle = 0;
C.C_M_0        = 0.09448;
C.C_M_alpha    = -0.6028;
C.C_M_q        = -15.56;
C.C_M_elevator = -1.921;
C.C_M_throttle = 0;

% longitudinal coefficents
C.C_Y_0        = -0.002226;
C.C_Y_beta     = -0.7678;
C.C_Y_p        = -0.1240;
C.C_Y_r        = 0.3666;
C.C_Y_aileron  = -0.02956;
C.C_Y_rudder   = 0.1158;
```

```
  C.C_L_0         = 0.0005910;
  C.C_L_beta      = -0.06180;
  C.C_L_p         = -0.5045;
  C.C_L_r         = 0.1695;
  C.C_L_aileron   = -0.09917;
  C.C_L_rudder    = 0.006934;
  C.C_N_0         = -0.003117;
  C.C_N_beta      = 0.006719;
  C.C_N_p         = -0.1585;
  C.C_N_r         = -0.1112;
  C.C_N_aileron   = -0.003872;
  C.C_N_rudder    = -0.08265;

  auto = my_autopilot_design((uavdata(p).v_max + uavdata(p).v_min)/2,0,C,Ts);
  function [sys,x0,str,ts] = environment(t,x,u,flag,env,P)
%environment
%  Simulate the environment, allowing for pop-up threats, moving
% threats and moving targets.
%
%
% modified 8/3/01 - Randy Beard

switch flag,

  %%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts] = mdlInitializeSizes(env,P);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
 case 2,
    sys = mdlUpdate(t,x,u,env,P);

  %%%%%%%%%%
  % Output %
  %%%%%%%%%%
 case 3,
  sys = mdlOutputs(t,x,u,env,P);

  %%%%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%%%
  case 9,
```

```
    sys = []; % do nothing

  %%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%
  otherwise
    error(['unhandled flag = ',num2str(flag)]);
end

%end dsfunc

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes(env,P)

  %input('In environment init');
  sizes = simsizes;
  sizes.NumContStates  = 0;
  sizes.NumDiscStates  = 1+2*size(env.threats,1)+2*size(env.popup,1);
  sizes.NumOutputs     = 1+2*size(env.threats,1)+2*size(env.popup,1);
  sizes.NumInputs      = 3*P;
  sizes.DirFeedthrough = 1;
  sizes.NumSampleTimes = 1;


  sys = simsizes(sizes);

  x0  = [size(env.threats,1); reshape(env.threats,2*size(env.threats,1),1); ...
     reshape(env.popup,2*size(env.popup,1),1)];
  str = [];
  ts  = [0 0];

% end mdlInitializeSizes
%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function xup = mdlUpdate(t,x,u,env,P)
```

```
  %input('In environment update');
  num_threats = x(1);
  num_popup   = size(env.threats,1)+size(env.popup,1)-num_threats;
  threats     = reshape(x(2:1+2*num_threats),num_threats,2);
  popup       = reshape(x(2+2*num_threats:length(x)),num_popup,2);
  for i=1:P,
    uav(:,i)    = u(1+3*(i-1):2+3*(i-1));
  end


  if num_popup>0,
    for i=1:P,
      % compute the closest pop-up threat to each uav
      tmp=repmat(uav([1:2],i),1,size(popup,1))-popup';
      dist = sqrt(diag(tmp'*tmp));
      [closest,index] = min(dist);

      % if a popup threat is close to the uav, then add that
      % threat to the threat list.
      if closest < 5,
        num_threats    = num_threats+1;
        threats        = [threats; popup(index,:)];
        popup          = [popup(1:index-1,:); popup(index+1:num_popup,:)];
        num_popup      = num_popup-1;
      end
    end
  end

  xup = [num_threats; reshape(threats,2*num_threats,1); ...
     reshape(popup,2*num_popup,1)];

%end mdlUpdate
%
%=============================================================================
% mdlOutputs
% Return Return the output vector for the S-function
%=============================================================================
%
function y = mdlOutputs(t,x,u,env,P)

  %input('In environment output');
  num_threats = x(1);
  threats     = x(2:1+2*num_threats);

  y = x;
```

```
%end mdlOutput


function [vp, vel] = generateVPpath(threats, st, tg, v)
    vp = fixVP(calcVP(threats, st, tg)); %FIX add smoother here
    [vp, vel] = fixPath(vp, v);


%
%
%=========================================================================
% fixVP
% This function removes nodes that have been added by the calcVP
% step, when required.  The calcVP module was written under the assumption
% that the airplane is not on a Voronoi point.  If it is, then the
% function adds nodes that shouldn't be there.  This function removes them.
%=========================================================================
function vp = fixVP(vp_in)

  n=size(vp_in,2);

  if 0,
    if norm(vp_in(2:3,1)-vp_in(2:3,2))<.01,
      vp = vp_in(:,2:n);
    else
      vp = vp_in;
    end
  end

  if norm(vp_in(2:3,1)-vp_in(2:3,3))<.01,
    vp = vp_in(:,3:n);
  else
    vp = vp_in;
  end

%
%
%
%=========================================================================
% calc_mult_VP
% Calculates a Voronoi-diagram-based path from the start point to the
% target avoiding threats
%=========================================================================
%
%function [vp,pvec] = calc_mult_VP(threats,st,tg);
```

```
function [vp,pvec] = calcVP(threats,st,tg);

tx = threats(:,1); ty = threats(:,2); [vx,vy] = voronoi(tx,ty);

nodes = find_nodes(vx,vy); [nm,nn] = size(nodes); nodes_aug =
[nodes(1,:) nn+1 nn+2; ...
         st(1) nodes(2,:) tg(1); ...
        st(2) nodes(3,:) tg(2)];

sn_ind = get_closest(nodes,st); tn_ind = get_closest(nodes,tg);

% Calculate a threat cost for each edge of the Voronoi diagram
threat_cost =
calc_threat_cost(vx,vy,tx,ty,nodes,st,sn_ind,tg,tn_ind);

%Kt is the priority between a safe path and a long path
%the number is 0<Kt<1.  A number closer to 1 implies a long, safe path
%while a number closer to 0 implies a shorter, more risky path
Kt=.3;
%Kt=0.001;

A = gen_cost_mat(vx,vy,nodes,threat_cost,st,sn_ind,tg,tn_ind,Kt);

%numVP is the number of paths
numVP = 1;
% Use bidirectional arcs.  Change cost matrix to add links in
% both directions.
[M,N] = size(A); D = -ones(M,N); for i=1:M,
   for j=1:N,
      if (A(i,j)~=Inf)& (A(i,j)~=0),
       D(i,j) = A(i,j);
       D(j,i) = A(i,j);
      end
   end
end D(M,:)=-1;

% Bidirectional arcs graph search called using D matrix
[VP,CostVP] = multipathepp(numVP,1,nn+2,D); [tmp,nVP] = size(VP);

% Bidirectional arcs graph search called using D matrix
[VP,CostVP] = multipathepp(numVP,1,nn+2,D); [tmp,nVP] = size(VP);

[tmp,pvec] = min(abs(VP-(nn+2))); nl = []; for i = 1:nVP,
   nl_tmp = VP(1:pvec(i),i)';
   nl = [nl nl_tmp];
end
```

```
vp = nodes_aug(:,nl);

if 0, numVP = 1;
% Use bidirectional arcs.  Change cost matrix to add links in
% both directions.
[M,N] = size(A); D = -ones(M,N); for i=1:M,
   for j=1:N,
      if (A(i,j)~=Inf)& (A(i,j)~=0),
       D(i,j) = A(i,j);
       D(j,i) = A(i,j);
      end
   end
end D(M,:)=-1;

% Unidirectional arcs graph search called using D matrix
[VP,CostVP] = MultiPath(numVP,1,nn+2,D);
% Unidirectional arcs graph search called using A matrix
% This .m-file is saved as MultiPath.m.SAVE
%[VP,CostVP] = MultiPath(numVP,1,nn+2,A);
[tmp,nVP] = size(VP);

[tmp,pvec] = min(abs(VP-1)); nl = []; for i = 1:nVP,
   nl_tmp = VP(pvec(i):-1:1,i)';
   nl = [nl nl_tmp];
end

vp = nodes_aug(:,nl); end
%
%=========================================================================
% find_nodes
%  from the description of the voronoi edges, find and number the nodes
%=========================================================================
%
function nodes = find_nodes(vx,vy);

vxx = [vx(1,:) vx(2,:)]; vyy = [vy(1,:) vy(2,:)];

[vyy_sort,i_sort] = sort(vyy);

tmpy = vyy_sort(1); tmpx = vxx(i_sort(1)); ind = 1; num = 1;

% changed Aug27,2002 so as not to eliminate nodes based only on y-values %
for i=2:length(vyy),
  if vyy_sort(i) > tmpy,
    ind = [ind i];
```

```
    tmpy = vyy_sort(i);
    tmpx = vxx(i_sort(i));
  else
    eqFlag = 0;
    for j=1:length(tmpx),
      if vxx(i_sort(i)) == tmpx(j),
        eqFlag = 1;
      end
    end
    if eqFlag == 0,
      ind = [ind i];
      tmpy = vyy_sort(i);
      tmpx = [tmpx vxx(i_sort(i))];
    end
  end
end

nodes = [1:1:length(ind); ...
         vxx(i_sort(ind)); ...
         vyy(i_sort(ind))];
%
%=========================================================================
% get_closest
%  calculate the closest three voronoi nodes to the specified point
%=========================================================================
%
function ind = get_closest(nodes,pt);

ni = nodes(1,:); nx = nodes(2,:); ny = nodes(3,:); px = pt(1); py
= pt(2);

delx = nx - px; dely = ny - py; dist_sq = delx.^2 + dely.^2;
[dist_sort,i_sort] = sort(dist_sq);

ind = i_sort(1:3);
%
%
%=========================================================================
% calc_threat_cost
% given the voronoi edges and the threat locations calculate
% the cost of traveling each edge
% cost = 1/d^4 * edge length where d is the distance from the threat
%        to the midpoint of the edge
%        1/6 point and 5/6 point also included
%
% last 6 rows of threat_cost correspond to costs from start and target
```

```
% to three verticies of Voronoi diagram
%=====================================================================
%
function threat_cost =
calc_threat_cost(vx,vy,tx,ty,nodes,spt,sn_i,tpt,tn_i);

bx = vx(1,:) + (vx(2,:)-vx(1,:))/6; by = vy(1,:) +
(vy(2,:)-vy(1,:))/6; mx = (vx(1,:)+vx(2,:))/2; my =
(vy(1,:)+vy(2,:))/2; ex = vx(1,:) + 5*(vx(2,:)-vx(1,:))/6; ey =
vy(1,:) + 5*(vy(2,:)-vy(1,:))/6;

L = sqrt((vx(1,:)-vx(2,:)).^2+(vy(1,:)-vy(2,:)).^2);

[m,n] = size(vx); p = length(tx);

for i=1:n,
  for j=1:p,
    threat_b(i,j) = 1/(.1*((bx(i)-tx(j))^2+(by(i)-ty(j))^2)^2)*L(i);
    threat_m(i,j) = 1/(.1*((mx(i)-tx(j))^2+(my(i)-ty(j))^2)^2)*L(i);
    threat_e(i,j) = 1/(.1*((ex(i)-tx(j))^2+(ey(i)-ty(j))^2)^2)*L(i);
  end;
end;

threat_mat = (threat_b + threat_m + threat_e)/3;

bxs = spt(1) + (nodes(2,sn_i)-spt(1))/6; bys = spt(2) +
(nodes(3,sn_i)-spt(2))/6; bxt = tpt(1) + (nodes(2,tn_i)-tpt(1))/6;
byt = tpt(2) + (nodes(3,tn_i)-tpt(2))/6; mxs =
(spt(1)+nodes(2,sn_i))/2; mys = (spt(2)+nodes(3,sn_i))/2; mxt =
(tpt(1)+nodes(2,tn_i))/2; myt = (tpt(2)+nodes(3,tn_i))/2; exs =
spt(1) + 5*(nodes(2,sn_i)-spt(1))/6; eys = spt(2) +
5*(nodes(3,sn_i)-spt(2))/6; ext = tpt(1) +
5*(nodes(2,tn_i)-tpt(1))/6; eyt = tpt(2) +
5*(nodes(3,tn_i)-tpt(2))/6;

Ls = sqrt((spt(1)-nodes(2,sn_i)).^2+(spt(2)-nodes(3,sn_i)).^2); Lt
= sqrt((tpt(1)-nodes(2,tn_i)).^2+(tpt(2)-nodes(3,tn_i)).^2); for
i=1:3,
  for j=1:p,
    threat_b_st(i,j) = 1/(.1*((bxs(i)-tx(j))^2+(bys(i)-ty(j))^2)^2)*Ls(i);
    threat_b_tg(i,j) = 1/(.1*((bxt(i)-tx(j))^2+(byt(i)-ty(j))^2)^2)*Lt(i);
    threat_m_st(i,j) = 1/(.1*((mxs(i)-tx(j))^2+(mys(i)-ty(j))^2)^2)*Ls(i);
    threat_m_tg(i,j) = 1/(.1*((mxt(i)-tx(j))^2+(myt(i)-ty(j))^2)^2)*Lt(i);
    threat_e_st(i,j) = 1/(.1*((exs(i)-tx(j))^2+(eys(i)-ty(j))^2)^2)*Ls(i);
    threat_e_tg(i,j) = 1/(.1*((ext(i)-tx(j))^2+(eyt(i)-ty(j))^2)^2)*Lt(i);
  end;
```

```
end;

threat_mat_start = (threat_b_st + threat_m_st + threat_e_st)/3;
threat_mat_target = (threat_b_tg + threat_m_tg + threat_e_tg)/3;

threat_cost = [sum(threat_mat,2)' sum(threat_mat_start,2)'
sum(threat_mat_target,2)'];
%
%=========================================================================
% gen_cost_mat
%  generate the cost adjacency matrix for the Voronoi diagram
%=========================================================================
%
function A = gen_cost_mat(vx,vy,nodes,tcost,spt,sni,tpt,tni,Ct);

Cf = 1-Ct;

ny = nodes(3,:);            % y coordinates of nodes
n = length(ny);
vy_csrt = sort(vy,1);       % sort y vertices by column
[dum,m] = size(vy_csrt);

tcostn = tcost(1:m); tcosts = tcost(m+1:m+3); tcostt =
tcost(m+4:m+6);

Af = inf*ones(n+2);         % create initial cost adjacency matrix
At=Af;
%At = inf*ones(n+2);
magf = 0; magt = 0;

for j = 1:n,
  Af(j,j) = 0;
  At(j,j) = 0;
  for i = 1:m,
    if abs(vy_csrt(1,i)-ny(j)) < eps,
      for k = 1:n,
        if abs(vy_csrt(2,i)-ny(k)) < eps,
          Af(j+1,k+1) = norm([vx(1,i)-vx(2,i);vy(1,i)-vy(2,i)]);
          At(j+1,k+1) = tcostn(i);
          magf = magf + Af(j+1,k+1);
          magt = magt + At(j+1,k+1);
        end
      end
    end
  end
end
```

```
Af(n+1,n+1) = 0; Af(n+2,n+2) = 0; At(n+1,n+1) = 0; At(n+2,n+2) =
0;

% Add the start points and end points into the cost matrix
for i=1:3,
  Af(1,sni(i)+1) = norm([spt(1)-nodes(2,sni(i));spt(2)-nodes(3,sni(i))]);
  Af(tni(i)+1,n+2) = norm([tpt(1)-nodes(2,tni(i));tpt(2)-nodes(3,tni(i))]);
  magf = Af(1,sni(i)+1) + Af(tni(i)+1,n+2);
  At(1,sni(i)+1) = tcosts(i);
  At(tni(i)+1,n+2) = tcostt(i);
  magt = At(1,sni(i)+1) + At(tni(i)+1,n+2);
end

Aff = Cf*Af/magf; Att = Ct*At/magt; A = Cf*Af/magf + Ct*At/magt;
%
%=========================================================================
% fixPaths
%  remove short path segments and adjust velocity
%=========================================================================
%
function [vp_new,vel_new] = fixPath(vp,vel);

  EPS = 50;

    % compute current transition time
    L = 0;
    for j=1:size(vp,2)-1,
      L = L + norm(vp([2:3],j+1)-vp([2:3],j));
    end
    T = L/vel;

    % remove close nodes
    vp2 = vp(:,1);
    for j = 1:size(vp,2)-2,
      if norm(vp2([2:3],end)-vp([2:3],j+1)) > EPS,
        vp2 = [vp2,vp(:,j+1)];
      else
        vp2(:,end) = (vp2(:,end)+vp(:,j+1))/2;
      end
    end
    if norm(vp2([2:3],end)-vp([2:3],end)) > EPS
      vp2 = [vp2,vp(:,end)];
    else
      vp2(:,end) = vp(:,end);
    end
```

```
    % remove angles close to 180 degrees
    vp_new = vp2(:,1);
    for j = 1:size(vp2,2)-2,
      wim1 = vp_new([2:3],end);
      wi   = vp2([2:3],j+1);
      wip1 = vp2([2:3],j+2);
      q1   = (wi-wim1)/norm(wi-wim1);
      q2   = (wip1-wi)/norm(wip1-wi);
      beta = acos(q1'*q2);
      if abs(beta) > pi/16,
        vp_new = [vp_new,vp2(:,j+1)];
      end
    end
    vp_new = [vp_new,vp2(:,end)];

    % compute new velocity to match transition time
    L = 0;
    for j=1:size(vp_new,2)-1,
      L = L + norm(vp_new([2:3],j+1)-vp_new([2:3],j));
    end
    vel_new = L/T;mex -c -DFOR_WINDOWS trajGen.cpp
mex -c -DFOR_WINDOWS rungkut.cpp mex -DFOR_WINDOWS traj_states.cpp
trajGen.obj rungkut.obj function auto = my_autopilot_design(u0,
th0, C, Ts);
  % u0  = nominal airspeed
  % th0 = nominal roll angle
  % C   = airplane coefficients

% General constants
g     = 9.8;                % gravitational constant (m/s^2)
S     = C.bw*(C.cw+C.dw)/4; % wing size, adjusted for taper and sweep (m^2)
q_dyn = 0.5*C.rho*u0^2;     % dynamic pressure
b     = C.b;                % wingspan (m)
cbar  = C.mean_chord;       % mean chord (m)
m     = C.m;                % mass (kg)
Ix    = C.J(1,1);           % moment of inertia about x-axis (kg-m)
Iy    = C.J(2,2);           % moment of inertia about y-axis (kg-m)
Iz    = C.J(3,3);           % moment of inertial about z-axis (kg-m)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% lateral mode
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % lateral directional derivatives (dimensional)
```

```
Y_beta    = q_dyn*S*C.C_Y_beta/m;
Y_p       = q_dyn*S*b*C.C_Y_p/2/m/u0;
Y_r       = q_dyn*S*b*C.C_Y_r/2/m/u0;
Y_rudder  = q_dyn*S*C.C_Y_rudder/m;
L_beta    = q_dyn*S*b*C.C_L_beta/Ix;
L_p       = q_dyn*S*b^2*C.C_L_p/2/Ix/u0;
L_r       = q_dyn*S*b^2*C.C_L_r/2/Ix/u0;
L_aileron = q_dyn*S*b*C.C_L_aileron/Ix;
L_rudder  = q_dyn*S*b*C.C_L_rudder/Ix;
N_beta    = q_dyn*S*b*C.C_N_beta/Iz;
N_p       = q_dyn*S*b^2*C.C_N_p/2/Iz/u0;
N_r       = q_dyn*S*b^2*C.C_N_r/2/Iz/u0;
N_aileron = q_dyn*S*b*C.C_N_aileron/Iz;
N_rudder  = q_dyn*S*b*C.C_N_rudder/Iz;

% state space equations for lateral mode
A_lat = [...
        Y_beta/u0, Y_p/u0, -(1-Y_r/u0), g*cos(th0)/u0;...
        L_beta, L_p, L_r, 0;...
        N_beta, N_p, N_r, 0;...
        0, 1, 0, 0;...
    ];
B_lat = [...
        0, Y_rudder/u0;...
        L_aileron, L_rudder;...
        N_aileron, N_rudder;...
        0, 0;...
     ];
 C_lat = eye(4);

 switch 2,
     case 1, % pole placement design
         a = 7;
         b = 14;
         poles = [-a-j*a, -a+j*a, -b-j*b, -b+j*b];
         K_lat = place(A_lat,B_lat,poles);
     case 2, % LQR
%          Q = diag([10000, 0.01, 0.01, 100]);
%          Q = diag([10000, 0.001, 0.001, 10]);
         Q = diag([100, 0, 0.001, 100]);
         R = diag([1,1]);
         K_lat = lqr(A_lat,B_lat,Q,R);
         L_lat = [];
     case 3, % LQG / LTR
         Q = diag([100, 0, 0.001, 100]);
         R = diag([1,1]);
```

```
        K_lat = lqr(A_lat,B_lat,Q,R);
        r = 100;
        Xi    = diag([.001, .001, .001, .001]) + r*B_lat*B_lat';
        Th    = diag([.01, .001, .001, .001]);
        L_lat = lqr(A_lat',C_lat',Xi,Th)';
        eig(A_lat-B_lat*K_lat-L_lat*C_lat)
        [auto.A_lat_d,auto.B_lat_d] = c2d(A_lat-B_lat*K_lat-L_lat*C_lat, L_lat,Ts);
end

auto.A_lat = A_lat;
auto.B_lat = B_lat;
auto.K_lat = K_lat;
auto.C_lat = C_lat;
auto.L_lat = L_lat;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% longitudinal mode
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % longitudinal directional derivatives (dimensional)
%  X_u        = 0;
  X_u        = q_dyn*S*C.C_X_0/m/u0;
  X_alpha    = q_dyn*S*C.C_X_alpha/m;
  X_w        = X_alpha/u0;
  X_elevator = q_dyn*S*C.C_X_elevator/m/u0;
%  Z_u        = 0;
  Z_u        = q_dyn*S*C.C_Z_0/m/u0;
  Z_alpha    = q_dyn*S*C.C_Z_alpha/m;
  Z_w        = Z_alpha/u0;
  Z_elevator = q_dyn*S*C.C_Z_elevator/m/u0;
%  M_u        = 0;
  M_u        = q_dyn*S*cbar*C.C_M_0/Iy/u0;
  M_alpha    = q_dyn*S*cbar*C.C_M_alpha/Iy;
  M_w        = M_alpha/u0;
  M_q        = q_dyn*S*cbar^2*C.C_M_q/2/u0/Iy;
  M_elevator = q_dyn*S*cbar*C.C_M_elevator/Iy;

  % state space equations for lateral mode
  A_lon = [...
          X_u, X_w, 0, -g*cos(th0);...
          Z_u, Z_w, u0, -g*sin(th0);...
          M_u, M_w, M_q, 0;...
          0, 0, 1, 0;...
      ];
  B_lon = [...
          X_elevator;...
```

```
        Z_elevator;...
        M_elevator;...
        0;...
    ];
C_lon = eye(4);

switch 2,
    case 1, % pole placement design
        a = 7;
        b = 14;
        poles = [-a-j*a, -a+j*a, -b-j*b, -b+j*b];
        K_lon = place(A_lon,B_lon,poles);
        L_lon = [];
    case 2, % LQR
        Q = diag([0.1,0.001,0.001,100]);
        R = 1;
        K_lon = lqr(A_lon,B_lon,Q,R);
        L_lon = [];
    case 3, % LQG / LTR
        Q = diag([0.1,0.001,0.001,1000]);
        R = 1;
        K_lon = lqr(A_lon,B_lon,Q,R);
        r = 100;
        Xi    = diag([.001, .001, .001, .001]) + r*B_lon*B_lon';
        Th    = diag([.01, .001, .001, .001]);
        L_lon = lqr(A_lon',C_lon',Xi,Th)';
        eig(A_lon-B_lon*K_lon-L_lon*C_lon)
        [auto.A_lon_d,auto.B_lon_d] =...
            c2d(A_lon-B_lon*K_lon-L_lon*C_lon, L_lon, Ts);
end

auto.A_lon = A_lon;
auto.B_lon = B_lon;
auto.C_lon = C_lon;
auto.K_lon = K_lon;
auto.L_lon = L_lon;


% parameter file for simulations
%
clear all startTime = 0.0; stopTime = 3*60.0; Ts = 1/100;

global P; global uavdata; global env; global M;

% number of vehicles
P = 1;
```

```
% initial configuration of uavs
%   fields are: uavdata(num).ic = [X0; Y0; psi0; V0; h0; hdot0]
%   where  (X0, Y0) = inertial position of aircraft
%          psi0     = heading angle in inertial coordinates
%          V0       = speed
%          h0       = altitude
%          hdot0    = climb rate
  %uavdata(1).ic = [-25;  -5;    pi/4;  1.5;  1;  0];
  uavdata(1).ic = [-500;  -100;    pi/4;  11;  1;  0];
  uavdata(2).ic = [ 25;  -5;  3*pi/4;  1;  1.1;  0];
  uavdata(3).ic = [ 25;  40; -3*pi/4;  1;  1.2;  0];
  uavdata(4).ic = [ -5;  40; -3*pi/4;  1;  1.3;  0];
  uavdata(5).ic = [-10;  -5;  3*pi/4;  1;  1.4;  0];

% target locations
  env.targets(:,:,1) = 20*[...
    20, 5;...
    10, 20;...
       -20, 25;...
       -10, 35;...
       -10, 0;...
        ];

% home locations (places the UAVs fly to after mission is completed
  env.home = 20*[...
      100, 0;...
      100, 10;...
      100, 20;...
      100, 30;...
      100, 40;...
      ];

% initially known threat locations
  env.threats = 20*[...
       -1,   4;...
       12,   9;...
      -12,   8;...
      -18,  11;...
        8,  14;...
       18,  16;...
       -8,  17;...
      -11,  17;...
      -20,  18;...
       15,  20;...
        2,  23;...
```

```
      -12,  23;...
        6,  27;...
       13,  27;...
       -1,  28;...
       19,  29;...
      -13,  30;...
       12,  33;...
       -2,  35;...
        0,  12;...
        5,  12;...
       -7,  19;...
        5,  19;...
        7,  35;...
        9,   2;...
        0,  -5;...  % new threats
       17,  -3;...
       25,   5;...
       28,  12;...
       25,  25;...
       15,  44;...
        4,  47;...
      -15,  40;...
      -22,  32;...
      -28,  12;...
      -17,  -8
      ];

% pop up threats, only become visible when airplane is
% within a threshhold of the threat
  env.popup = 20*[...
    -15, -2;...
     -8,  5;...
%   22,  20;...
      ];


% airplane coefficients and control gains
  for i=1:P,
    %uavdata(i).psidot_max = 0.2375; % works!
    uavdata(i).psidot_max = 0.271;     % turning rate limit
    uavdata(i).v_max      = 11.5;      % 11;        % maximum plane velocity
    uavdata(i).v_min      = 9.4;       % 10;        % minimum plane velocity
    %uavdata(i).tau_psi    = 0.55; %works!
    uavdata(i).tau_psi    = 0.55;       %3.81;      % autopilot constant
    uavdata(i).tau_V      = 0.192;     %0.14;       % autopilot constant
    uavdata(i).tau_ha     = 3.5118;    % 2.426      % autopilot constant
```

```
    uavdata(i).tau_hb     = 0.8447;          % autopilot constant
    uavdata(i).size       = 0.08;            % scale on airplane drawing
    uavdata(i).k_hdot     = 10;              % control gain
    uavdata(i).k_h        = 10;              % control gain
    uavdata(i).k_x        = 1;               % control gain
    uavdata(i).k_y        = 1;               % control gain
    uavdata(i).k_z        = 1;               % control gain
    uavdata(i).turn_param = 1;               % 1=parameterized turn in
                                             % trajectory generator
    uavdata(i).equivdist  = 1;               % 1=trajectories have same
                                             % length as Voronoi path
    uavdata(i).kappa      = .01;             % parameter for trajectory
                                             % generator

    uavdata(i).K1         = 10*eye(2);       % control gain for ATT
    uavdata(i).K2         = 1*eye(2);        % control gain for ATT
    uavdata(i).gam        = 1/10;            % adaptation gain for ATT
    uavdata(i).bkstepK    = 10.0;
    uavdata(i).bkstepG    = 1.0;

    % tracker parameters
    uavdata(i).track_gam0 = 0.5;
    uavdata(i).track_gam1 = 0.5;
    uavdata(i).track_gam2 = 0.5;
    uavdata(i).track_k1   = 2.0;
    uavdata(i).track_eps_vmin = 0.4;
    uavdata(i).track_eps_vmax = 1.5;
    %uavdata(i).track_eps_wmax = 0.05; %works!
    uavdata(i).track_eps_wmax = 0.1;
  end

  % number of storage elements for path
  M = 100;

  % calculate initial conditions
  sim_ic = calcInitialCond;

  dof12paramfunction [sys,x0,str,ts] = plotEnv(t,x,u,flag,plane,env,M,P)
%plotTrajectory - plot trajectory being planned.
%
%  Expected inputs are:
%    u(1)   - trajectories
%    u(2)   - waypoint paths
%    u(3)   - UAV states
%    u(4)   - target locations
%    u(5)   - threat locations
```

```
%
% modified 8/3/01  - RWB
%          9/26/02 - RWB

switch flag,

  %%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts] = mdlInitializeSizes(P,M);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys = mdlUpdate(t,x,u,plane,env,P,M);

  %%%%%%%%%%
  % Output %
  %%%%%%%%%%
  case 3,
    sys = [];

  %%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%
  case 9,
    sys = []; % do nothing

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['unhandled flag = ',num2str(flag)]);
end

%end dsfunc

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes(P,M)
```

```
sizes = simsizes; sizes.NumContStates  = 0; sizes.NumDiscStates  =
3+P*(1+1+1+1+1+3*M);
  % x(1)                    = state to indicate plot initialization
  % x(2)                    = num_threats_stored
  % x(3:3+P-1)              = figure handle for paths
  % x(3+P:3+2*P-1)          = figure handle for trajectories
  % x(3+2*P:3+3*P-1)        = figure handle for UAVs
  % x(3+3*P:3+4*P-1)        = figure handle for targets
  % x(3+4*P)                = figure handle for threats
  % x(4+4*P:4+4*P-1)        = num_waypoints_stored
  % x(4+4*P+1:4+4*P+P*3*M) = stored waypoint paths
sizes.NumOutputs     = 0; sizes.NumInputs      = -1;
sizes.DirFeedthrough = 1; sizes.NumSampleTimes = 1;


sys = simsizes(sizes);

x0 = zeros(sizes.NumDiscStates,1); str = []; ts = [0 0];
%ts  = [.05 0];
%ts = [.2 0];

% end mdlInitializeSizes
%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function xup = mdlUpdate(t,x,u,plane,env,P,M)

  initialize           = x(1);
  fig_threats          = x(2);
  fig_path             = x(3:3+P-1);
  fig_traj             = x(3+P:3+2*P-1);
  fig_uav              = x(3+2*P:3+3*P-1);
  fig_target           = x(3+3*P:3+4*P-1);
  num_threats_stored   = x(3+4*P);
  for i=1:P,
    num_waypoints_stored(i)   = x(4+4*P+(3*M+1)*(i-1));
    path_stored(:,i)          = x(5+4*P+(3*M+1)*(i-1):...
                4+4*P+(3*M+1)*i-1);
  end
  for i=1:P,
    traj(:,i)          = u(1+3*(i-1):3+3*(i-1));
```

```
end
for i=1:P,
  num_waypoints(i)    = u(1+3*P+(3*M+2)*(i-1));
  vel(i)              = u(2+3*P+(3*M+2)*(i-1));
  path(:,i)           = u(3+3*P+(3*M+2)*(i-1):...
            3+3*M-1+3*P+(3*M+2)*(i-1));
end
for i = 1:P,
  x_plane(:,i) = u(1+P*(5+3*M)+6*(i-1):...
           6+P*(5+3*M)+6*(i-1));
end
for i=1:P,
  target(:,i)  = u(1+2*(i-1)+P*(11+3*M):...
                   2+2*(i-1)+P*(11+3*M));
end
num_threats     = u(1+P*(13+3*M));
tmp             = u(2+P*(13+3*M):...
          1+2*num_threats+P*(13+3*M));
threats         = reshape(tmp,num_threats,2);


if initialize==0,
  % initialize the plot
  initialize = 0.05;
  figure(1), clf
  axis(20*[-10,50,-30,30]);
  grid on
  xlabel('Y (East)');
  ylabel('X (North)');
  hold on


  % plot threats
  fig_threats=plot(threats(:,2),threats(:,1),'.','EraseMode','none');
  zoom on

  % plot way-point path
  for i=1:P,
    tmp         = reshape(path(1:3*num_waypoints(i),i),3, ...
                num_waypoints(i));
    fig_path(i) = plot(tmp(2,:),tmp(1,:),'green','EraseMode','none');
  end

  % plot trajectory
  for i=1:P,
    fig_traj(i) = trajPlotInit(traj(1:2,i),'red','xor');
```

```
  end

  % plot UAVs
  for i=1:P,
    fig_uav(i) = planePlotInit(x_plane(1:3,i),'blue',plane.size,'xor');
  end

  % plot targets
  for i=1:P,
    fig_target(i) = plot(target(2,i),target(1,i),'s');
  end

else  % do this at every time step
  % make sure we only do this every 20th of a second
  if(t >= initialize),
     initialize = t + 0.05;
     % plot threats
     %figure(3)
     if num_threats~=num_threats_stored,
        set(fig_threats,'XData',threats(:,2),'YData',threats(:,1));
        drawnow
     end

     % plot way-point path
     for i=1:P,
        if norm(path(:,i)-path_stored(:,i))~=0,
           tmp      = reshape(path(1:3*num_waypoints(i),i),3,num_waypoints(i));
           set(fig_path(i),'XData',tmp(2,:),'YData',tmp(1,:));
           drawnow
        end
     end

     % plot trajectory
     for i=1:P,
        trajPlot(fig_traj(i),traj(1:2,i));
     end

     % plot airplanes
     for i=1:P
        planePlot(fig_uav(i),x_plane(1:3,i),'blue',plane.size);
     end

     % plot targets
     for i=1:P,
        set(fig_target(i),'XData',target(2,i),'YData',target(1,i));
     end
```

```
    end;
  end

  xup = [initialize; fig_threats; fig_path; fig_traj;...
     fig_uav; fig_target; num_threats_stored];
  for i=1:P,
    xup = [xup;num_waypoints_stored(i);...
       reshape(path_stored(:,i),3*M,1)];
  end

%end mdlUpdate

%=============================================================================
%=============================================================================
% User defined functions
%=============================================================================
%=============================================================================

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function handle=planePlotInit(y,color,APSIZE,mode);
% planePlotInit:  plot plane at configuration y with color
  ap = apTranslate(apRotate(20*APSIZE*apData,y(3)),y(1:2));
  handle=plot(ap(2,:),ap(1,:),color,'EraseMode',mode);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function planePlot(handle,y,color,APSIZE);
% planePlot:  plot plane at the configuration given by y.
  ap = apTranslate(apRotate(20*APSIZE*apData,y(3)),y(1:2));
  set(handle,'XData',ap(2,:),'YData',ap(1,:));
  drawnow

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function handle=trajPlotInit(y,color,mode);
% trajPlotInit:  plot desired trajectory position
  th = 0:2*pi/10:2*pi;
  rab = 20*5*[y(1)+cos(th); y(2)+sin(th)];
  handle=plot(rab(2,:),rab(1,:),color,'EraseMode',mode);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function trajPlot(handle,y);
% trajPlot:  plot desired trajectory position
  th = 0:2*pi/10:2*pi;
  R  = 20*(.5);
  rab = [y(1)+R*cos(th); y(2)+R*sin(th)];
  set(handle,'XData',rab(2,:),'YData',rab(1,:));
  drawnow
```

```
%%%%%%%%%%%%%%%%%%%%%%%
function apOut = apRotate(ap,phi);
% apRotate:  rotate an airplane defined by ap by the angle phi
  R     = [cos(phi) -sin(phi); sin(phi) cos(phi)];
  apOut = R*ap;

%%%%%%%%%%%%%%%%%%%%%%%
function apOut = apTranslate(ap,r);
% apTranslate:  translate the airplane by the vector r
  apOut = ap + r*ones(1,length(ap));

%%%%%%%%%%%%%%%%%%%%%%%
function ap = apData;
% apData:  define the points on the aircraft
  ap = [...
    5,0;...
    -5,-5;...
    -2,0;...
    -5,5;...
    5,0;...
  ]';



function [sys,x0,str,ts] =
targetMgr(t,x,u,flag,uavdata,env,M,P,xx0)
%  Target Manager
%
%  Modified 3/26/02 - RB

switch flag,

  %%%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts] = mdlInitializeSizes(uavdata,env,M,P,xx0);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys = mdlUpdate(t,x,u,uavdata,env,M,P);

  %%%%%%%%%%
```

```
  % Output %
  %%%%%%%%%%
  case 3,
    sys = mdlOutputs(t,x,u,uavdata,env,M,P);

  %%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%
  case 9,
    sys = []; % do nothing

  %%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%
  otherwise
    error(['unhandled flag = ',num2str(flag)]);
end

%end dsfunc

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes(uavdata,env,M,P,xx0)

  %input('In targetmgr init');
  sizes = simsizes;
  sizes.NumContStates  = 0;
  sizes.NumDiscStates  = P + 2*P + P;
    %   for i=1:P,
    %     state(i)           = x(i);
    %     target(:,i)        = x(1+2*(i-1)+P:2+2*(i-1)+P);
    %     target_ptr(i)      = x(i+3*P);
    %   end
   sizes.NumOutputs     = 2*P;
    % y             = targets
  sizes.NumInputs      = P;
    %  for i=1:P,
    %    new_path_flag(i)   = u(i);
    %  end
  sizes.DirFeedthrough = 1;
  sizes.NumSampleTimes = 1;
```

```
  sys = simsizes(sizes);

  % initialize state
  x0 = xx0; % done in calcInitialCond

  str = [];
  ts  = [0 0];

% end mdlInitializeSizes
%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function xup = mdlUpdate(t,x,u,uavdata,env,M,P)

  %input('In targetmgr update');
  for i=1:P,
    state(i)           = x(i);
    target(:,i)        = x(1+2*(i-1)+P:2+2*(i-1)+P);
    target_ptr(i)      = x(i+3*P);
    new_target_flag(i) = u(i);
  end

  for i=1:P,

    switch state(i),
     case 1, % loop here until a path is requested
      if new_target_flag(i)==1,
        state(i) = 2;
      else
        state(i) = 1;
      end

     case 2, % output the next target
      if target_ptr(i)+1 > size(env.targets(:,:,i),1),
        target_ptr(i) = 1;
      else
        target_ptr(i) = target_ptr(i)+1;
      end
      target(:,i) = env.targets(target_ptr(i),:,i)';
      state(i) = 3;

     case 3, % wait for confirmation of received target
```

```
      if new_target_flag(i)==0,
        state(i) = 1;
      else
        state(i) = 3;
      end

     otherwise
      disp('Unknown state')
    end
  end

  xup = [...
        state;...
        reshape(target(:,i),2*P,1);...
        target_ptr;...
    ];

%end mdlUpdate
%
%=============================================================================
% mdlOutputs
% Return Return the output vector for the S-function
%=============================================================================
%
function y = mdlOutputs(t,x,u,uavdata,env,M,P)

  %input('In targetmgr output');
  for i=1:P,
    state(i)           = x(i);
    target(:,i)        = x(1+2*(i-1)+P:2+2*(i-1)+P);
    target_ptr(i)      = x(i+3*P);
    new_target_flag(i) = u(i);
  end

  % output the paths for each UAV
  y = reshape(target,2*P,1);

%end mdlUpdate

function [sys,x0,str,ts] = att(t,x,u,flag,uavdata,P)
% s-function implementation of UAV dynamics (assuming an autopilot).
%
% Modified: 9/26/02 - RWB
%

switch flag,
```

```
  %%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes(uavdata,P);

  %%%%%%%%%%%%%%%%
  % Derivatives %
  %%%%%%%%%%%%%%%%
  case 1,
    sys=mdlDerivatives(t,x,u,uavdata,P);

  %%%%%%%%%%
  % Outputs %
  %%%%%%%%%%
  case 3,
    sys=mdlOutputs(t,x,u,uavdata,P);

  %%%%%%%%%%%%%%%%%%
  % Unhandled flags %
  %%%%%%%%%%%%%%%%%%
  case { 2, 4, 9 },
    sys = [];

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end
% end csfunc

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes(uavdata,P)

sizes = simsizes; sizes.NumContStates  = 6*P; sizes.NumDiscStates
= 0; sizes.NumOutputs     = 6*P; sizes.NumInputs      = 3*P;
sizes.DirFeedthrough = 0; sizes.NumSampleTimes = 1;
```

```
sys = simsizes(sizes);

%uavdata.ic: x, y, psi, v, h, hdot

%x0 = uavdata(1).ic+[1.5; 1; 0.8; 0; 0; 0];
x0 = uavdata(1).ic;     %adding some initial tracking errors
for p=2:P,
    x0 = [x0; uavdata(p).ic];
end str = []; ts  = [0 0];

% end mdlInitializeSizes
%
%=============================================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=============================================================================
%
function xdot=mdlDerivatives(t,x,u,uavdata,P)

% interpret inputs
for p=1:P,
    X(p)     = x(1+6*(p-1));
    Y(p)     = x(2+6*(p-1));
    psi(p)   = x(3+6*(p-1));
    v(p)     = x(4+6*(p-1));
    h(p)     = x(5+6*(p-1));
    hdot(p)  = x(6+6*(p-1));
    psi_c(p) = u(1+3*(p-1));
    v_c(p)   = u(2+3*(p-1));
    h_c(p)   = u(3+3*(p-1));
end

% uav dynamics
for p=1:P,
    xdot(1+6*(p-1)) = v(p)*cos(psi(p));
    xdot(2+6*(p-1)) = v(p)*sin(psi(p));
    xdot(3+6*(p-1)) = uavdata(p).tau_psi*(psi_c(p)-psi(p));

    % saturate velocity
    vdot_temp = uavdata(p).tau_V*(v_c(p)-v(p));

    %if (v(p) >= uavdata(p).v_max)&(vdot_temp>0),
    %  xdot(4+6*(p-1)) = 0;
    %elseif (v(p) <= uavdata(p).v_min)&(vdot_temp<0),
    %  xdot(4+6*(p-1)) = 0;
    %else
```

```
      xdot(4+6*(p-1)) = vdot_temp;
    %end

    xdot(5+6*(p-1)) = hdot(p);
    xdot(6+6*(p-1)) = -uavdata(p).tau_hb*hdot(p) + uavdata(p).tau_ha*(h_c(p)-h(p));
end

% end mdlDerivatives
%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function y=mdlOutputs(t,x,u,uavdata,P)

% output the states
y =  x;

% end mdlOutputs

function y = uav_linear_forces(u,C)
% uavforces
%
% forces acting in the MAGICC lab UAV
%
% Modification History:
%   1/11/03 - RWB
%

  % inputs
  % z         = [u(1); u(2)];  % inertial position of UAV
  % h         = u(3);          % altitude
  V         = u(4);          % air speed
  alpha     = u(5);          % angle of attack
  beta      = u(6);          % sideslip angle
  phi       = u(7);          % roll
  theta     = u(8);          % pitch
  psi       = u(9);          % yaw
  p         = u(10);         % roll rate
  q         = u(11);         % pitch rate
  r         = u(12);         % yaw rate
  delta_er  = u(13);         % deflection of right elevon
  delta_el  = u(14);         % deflection of left elevon
  delta_rr = u(15);         % deflection of right rudder
  delta_rl = u(16);         % deflection of left rudder
```

```
delta_t   = u(17);          % engine input

rudder    = (delta_rr+delta_rl)/2;
elevator  = (delta_er+delta_el)/2;
aileron   = (delta_er-delta_el)/2;
throttle  = delta_t;

% forces due to gravity
F_gravity = [...
      -C.m*C.g*sin(theta);...
      C.m*C.g*cos(theta)*sin(phi);...
      C.m*C.g*cos(theta)*cos(phi)];

cbar = C.mean_chord;
b    = C.wingspan;

% longitudinal Force and Moment coefficients
C_X = C.C_X_0...
    + C.C_X_alpha * alpha...
    + C.C_X_q * (cbar/V) * q...
    + C.C_X_elevator * elevator...
    + C.C_X_throttle * throttle;
C_Z = C.C_Z_0...
    + C.C_Z_alpha * alpha...
    + C.C_Z_q * (cbar/V) * q...
    + C.C_Z_elevator * elevator...
    + C.C_Z_throttle * throttle;
C_M = C.C_M_0...
    + C.C_M_alpha * alpha...
    + C.C_M_q * (cbar/V) * q...
    + C.C_M_elevator * elevator...
    + C.C_M_throttle * throttle;

% lateral Force and Moment coefficients
C_Y = C.C_Y_0...
    + C.C_Y_beta * beta...
    + C.C_Y_p * (b/2/V) * p...
    + C.C_Y_r * (b/2/V) * r...
    + C.C_Y_aileron * aileron...
    + C.C_Y_rudder * rudder;
C_L = C.C_L_0...
    + C.C_L_beta * beta...
    + C.C_L_p * (b/2/V) * p...
    + C.C_L_r * (b/2/V) * r...
    + C.C_L_aileron * aileron...
    + C.C_L_rudder * rudder;
```

```
  C_N = C.C_N_0...
      + C.C_N_beta * beta...
      + C.C_N_p * (b/2/V) * p...
      + C.C_N_r * (b/2/V) * r...
      + C.C_N_aileron * aileron...
      + C.C_N_rudder * rudder;


  % wing size, adjusted for taper and sweep
  S = C.bw*(C.cw+C.dw)/4;

  % dynamic pressure
  q_dyn = 0.5*C.rho*V^2;

  % aerodynamic forces and moments
  F_aero = q_dyn * S * [ C_X; C_Y; C_Z];
  T_aero = q_dyn * S * [ (b/2) * C_L; cbar * C_M; (b/2) * C_N];


  % output total forces and torques
  F = F_gravity + F_aero;
  T = T_aero;

  % create output
  y = [F; T];
function xdot = uavdynamics(u,C)
% uavdynamics
%
% specify dynamics of the MAGICC lab UAV
%
% Modification History
%   12/6/02 - RWB
%

  m = C.m;
  J = C.J;

  X      = u(1);  % inertial X coordinate (north)
  Y      = u(2);  % inertial Y coordinate (East)
  h      = u(3);  % altitude in inertial coordinates
  V      = u(4);  % Airspeed
  alpha  = u(5);  % angle of attack
  beta   = u(6);  % sideslip angle
  phi    = u(7);  % roll angle
  theta  = u(8);  % pitch angle
  psi    = u(9);  % yaw angle
```

```
  p      = u(10); % roll rate
  q      = u(11); % pitch rate
  r      = u(12); % yaw rate

  F     = u(13:15);   % forces in body frame
  T     = u(16:18);   % torques in body frame

  % rotation matrices
  R_roll = [...
          1, 0, 0;...
          0, cos(phi), sin(phi);...
          0, -sin(phi), cos(phi)];
  R_pitch = [...
          cos(theta), 0, sin(theta);...
          0, 1, 0;...
          -sin(theta), 0, cos(theta)];
  R_yaw = [...
          cos(psi), sin(psi), 0;...
          -sin(psi), cos(psi), 0;...
          0, 0, 1];

  % compute body axis velocities
  u  = V*cos(alpha)*cos(beta);  % body velocity along x axis
  v  = V*sin(beta);             % body velocity along y axis
  w  = V*sin(alpha)*cos(beta);   % body velocity along z axis

  % compute xdot
  X_Y_h_dot = R_yaw'*R_pitch'*R_roll'*[u; v; -w];
  Vdot      = [cos(alpha)*cos(beta); sin(beta); sin(alpha)*cos(beta)]'*F/m;
  alphadot  = (1/V/cos(beta))*(1/m*(-F(1)*sin(alpha)+F(3)*cos(alpha)))...
              +q -(p*cos(alpha)+r*sin(alpha))*tan(beta);
  betadot   = (1/V/m)*(-F(1)*cos(alpha)*sin(beta) + F(2)*cos(beta)...
              - F(3)*sin(alpha)*sin(beta)) + p*sin(alpha) - r*cos(alpha);
  Edot     = inv([1, 0, -sin(theta);...
                  0, cos(phi), sin(phi)*cos(theta);...
                  0, -sin(phi), cos(phi)*cos(theta)])*[p;q;r];
  Wdot     = inv(J)*(-cross([p;q;r],J*[p;q;r]) + T);

  xdot = [X_Y_h_dot; Vdot; alphadot; betadot; Edot; Wdot];

function [sys,x0,str,ts] = wayPointMng(t,x,u,flag,plane,M,env,xx0)
%WayPointMng
%  Way point manager - feeds the trajectory generator N way points
%                      at a time.
%
%  Modified 3/26/02 - RB
```

```
  N=3;   % number of way points passed to trajectory generator

switch flag,

  %%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts] = mdlInitializeSizes(N,M,plane,env,xx0);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys = mdlUpdate(t,x,u,N,M,plane,env);

  %%%%%%%%%%
  % Output %
  %%%%%%%%%%
  case 3,
    sys = mdlOutputs(t,x,u,N,M,plane,env);

  %%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%
  case 9,
    sys = []; % do nothing

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['unhandled flag = ',num2str(flag)]);
end

%end dsfunc

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes(N,M,plane,env,xx0)
```

```
  %input('In waypointmng init');
  sizes = simsizes;
  sizes.NumContStates  = 0;
  sizes.NumDiscStates  = 1+1+3+1+6*M+1+1+1+3*M+1;
    %   state            = x(1);
    %   request_new_path = x(2);
    %   w0               = x(3:5);
    %   wpPtr            = x(5);
    %   wpQueue          = x(6:5+6*M);
    %   num_threats_prev = x(6+6*M);
    %   counter          = x(7+6*M);
    %   path_len         = x(8+6*M);
  sizes.NumOutputs     = 1+3+1+9;
    % y(1)     = request_new_path;
    % y(2)     = w0;
    % y(3)     = new_vel;
    % y(4)     = wpQueue(1:9)
  sizes.NumInputs      = -1;
    %   new_path_len     = u(1);
    %   new_vel          = u(2);
    %   new_path         = u(3:2+3*M);
    %   new_wp_flag      = u(3+3*M);
    %   traj             = u(4+3*M:6+3*M);
    %   num_threats      = u(7+3*M);
    %   threats          = reshape(u(8+3*M:7+3*M+2*num_threats),num_threats,2);

  sizes.DirFeedthrough = 1;
  sizes.NumSampleTimes = 1;

  sys = simsizes(sizes);

  x0 = xx0;

  str = [];
  ts  = [0 0];

% end mdlInitializeSizes
%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function xup = mdlUpdate(t,x,u,N,M,plane,env)
```

```
%input('In waypointmng update');
state            = x(1);
request_new_path = x(2);
w0               = x(3:5);
wpPtr            = x(6);
wpQueue          = x(7:6+6*M);
num_threats_prev = x(7+6*M);
counter          = x(8+6*M);
path_len         = x(9+6*M);
stored_path      = x(10+6*M:9+9*M);
stored_path_len  = x(10+9*M);
path             = wpQueue(1:3*path_len);
new_path_len     = u(1);
new_vel          = u(2);
new_path         = u(3:2+3*new_path_len);
new_wp_flag      = u(3+3*M);
traj             = u(4+3*M:6+3*M);
num_threats      = u(7+3*M);
threats          = reshape(u(8+3*M:7+3*M+2*num_threats),num_threats,2);

%%input('in way point manager update');
%if t >= 46.50,
%  t
%  state
%  wpPtr
%  wpQueue(wpPtr:wpPtr + 11)
%  path_len
%  input('top of update');
%end

switch state,

 case 1,  %state 1 loops here until pop-threat OR changed path OR request flag from traj.
   if num_threats~=num_threats_prev,
     state = 6;
   elseif (new_path_len~=stored_path_len) | ...
  (norm(stored_path(1:3*stored_path_len)-new_path)~=0),
     state = 5;
   elseif new_wp_flag==1,
     state = 2;
   else
     state = 1;
   end

   case 2,  %got a request flag from traj_gen
        %check for pop-up threats
```

```
      %try to send next wp
      %if 3 or less wps are in the Q, go to state 4
      %otherwise go to state 3
  wpPtr = wpPtr+3;

  if num_threats~=num_threats_prev,
    state = 6;
  elseif (path_len-(wpPtr-1)/3) <= 3,
    state = 4;
  else
    state = 3;
  end

  case 3,   %state 3 loops until pop-threat OR request flag from traj_gen has gone down,
  if num_threats~=num_threats_prev,
    state = 6;
  elseif new_wp_flag==0,
    state = 1;
  else
    state = 3;
  end

case 4,  %state 4 is when there are not enough wps in Q
  request_new_path = 1;
  w0 = wpQueue(3*path_len-2:3*path_len); %set w0 to last wp in Q

  if num_threats~=num_threats_prev,  %check for a pop-up threat
    state = 6;
  else
    state = 5;                       %otherwise go to state 5
  end

case 5,  %wait here for the path to change because we ran out of wps in our Q
  if num_threats~=num_threats_prev,  %check for a pop-up threat
    state = 6;
  elseif (new_path_len~=stored_path_len) | ...              %check if the path has chang
 (norm(stored_path(1:3*stored_path_len)-new_path)~=0),
    request_new_path = 0;                                   %the path has changed so we
    wpQueue(1:(3*(path_len-1)-wpPtr+1)) = wpQueue(wpPtr:3*(path_len-1));
    wpQueue(3*(path_len-1)-wpPtr+2:3*(path_len-1)-wpPtr+3*new_path_len+1)...
     = new_path(1:3*new_path_len);
    path_len = new_path_len+((path_len-1) - (wpPtr-1)/3);
    wpPtr = 1;
    stored_path = [new_path(1:3*new_path_len);...
          zeros(3*M-3*new_path_len,1)];
    stored_path_len = new_path_len;
```

```
    if path_len <= 3,   %ran out of wps again
      state = 4;
    else
      state = 1;
    end
  else
    state = 5;
  end

case 6,  %pop-up threat
  request_new_path = 2;
  num_threats_prev = num_threats;
  w0 = traj;
  state = 7;

case 7,  %pop-up threat cont - loop here until path changes
  if (new_path_len~=stored_path_len) | ...
 (norm(stored_path(1:3*stored_path_len)-new_path)~=0),
    request_new_path = 0;
    wpQueue(1:3*new_path_len) = new_path(1:3*new_path_len);
    path_len = new_path_len;
    wpPtr = 1;
    stored_path = [new_path(1:3*new_path_len);...
           zeros(3*M-3*new_path_len,1)];
    stored_path_len = new_path_len;

    if stored_path_len <= 3,
      state = 4;
    else
      state = 1;
    end
  else
    state = 7;
  end

 otherwise
  disp('Unknown state')
end

xup = [...
  state;...
  request_new_path;...
  w0;...
  wpPtr;...
  wpQueue;...
```

```
    num_threats_prev;...
    counter;...
    path_len;...
    stored_path;...
    stored_path_len;...
    ];

%end mdlUpdate
%
%=============================================================================
% mdlOutputs
% Return Return the output vector for the S-function
%=============================================================================
%
function y = mdlOutputs(t,x,u,N,M,plane,env)

  %input('In waypointmng output');
  state            = x(1);
  request_new_path = x(2);
  w0               = x(3:5);
  wpPtr            = x(6);
  wpQueue          = x(7:6+6*M);
  num_threats_prev = x(7+6*M);
  counter          = x(8+6*M);
  path_len         = x(9+6*M);
  path             = wpQueue(1:3*path_len);
  new_path_len     = u(1);
  new_vel          = u(2);
  new_path         = u(3:2+3*new_path_len);
  new_wp_flag      = u(3+3*M);
  traj             = u(4+3*M:6+3*M);
  num_threats      = u(7+3*M);
  threats          = reshape(u(8+3*M:7+3*M+2*num_threats),num_threats,2);

  %input('in way point manager output');
  y = [request_new_path; w0; new_vel; wpQueue(wpPtr:wpPtr+8)];

%end mdlUpdate
function [sys,x0,str,ts] = wpp(t,x,u,flag,uavdata,env,M,P,xx0)
%Waypoint Path Planner
%
%  Modified 3/26/02 - RB

switch flag,

  %%%%%%%%%%%%%%%%%%
```

```
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts] = mdlInitializeSizes(uavdata,env,M,P,xx0);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys = mdlUpdate(t,x,u,uavdata,env,M,P);

  %%%%%%%%%%
  % Output %
  %%%%%%%%%%
  case 3,
    sys = mdlOutputs(t,x,u,uavdata,env,M,P);

  %%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%
  case 9,
    sys = []; % do nothing

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['unhandled flag = ',num2str(flag)]);
end

%end dsfunc

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes(uavdata,env,M,P,xx0)

  %input('In wpp init');
  sizes = simsizes;
  sizes.NumContStates  = 0;
  sizes.NumDiscStates  = 1 + P + (2+3*M)*P + 2*P + P;
    %  num_threats_prev     = x(1);
    %   for i=1:P,
```

```
    %     state(i)           = x(i+1);
    %     path(:,i)          = x(1+(2+3*M)*(i-1)+P:(2+3*M)*i-1+P);
    %     path(:,i) = [num_waypoints; velocity; waypoints padded with
    %                  zeros];
    %     target_stored(:,i) =
    %                       x(1+2*(i-1)+P*(3+3*M):2+2*(i-1)+P*(3+3*M));
    %     new_target_flag(i) = x(i+1+P+(2+3*M)*P+2*P)
    %   end
   sizes.NumOutputs     = P + (2+3*M)*P;
    % y(1:P)             = flag requesting new target
    % y(P+1:end)         = paths
    %     path(:,i) = [num_waypoints; velocity; waypoints padded with
    %                  zeros];
  sizes.NumInputs      = 2*P + 4*P + 1 + 2*size(env.threats,1)...
                         + 2*size(env.popup,1);
    %    for i=1:P,
    %      target(:,i)        = u(1+2*(i-1) : 2+2*(i-1));
    %      new_path_flag(i)   = u(1+4*(i-1)+2*P);
    %      traj(:,i)          = u(2+4*(i-1)+2*P:4+4*(i-1)+2*P);
    %    end
    %    num_threats          = u(1+3*P);
    %    threats              = reshape(u(2+6*P:1+2*num_threats+6*P), ...
    %                     num_threats,2);
  sizes.DirFeedthrough = 1;
  sizes.NumSampleTimes = 1;

  sys = simsizes(sizes);

  % initialize state
  x0 = xx0;

  str = [];
  ts  = [0 0];

% end mdlInitializeSizes
%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function xup = mdlUpdate(t,x,u,uavdata,env,M,P)

  %input('In wpp update');
  num_threats_prev     = x(1);
```

```
for i=1:P,
  state(i)           = x(i+1);
  path(:,i)          = x(1+(2+3*M)*(i-1)+P+1:(2+3*M)*i+P+1);
  target_stored(:,i) = x(1+2*(i-1)+P*(3+3*M)+1:2+2*(i-1)+P*(3+3*M)+1);
  new_target_flag(i) = x(i+1+P+(2+3*M)*P+2*P);
  target(:,i)        = u(1+2*(i-1) : 2+2*(i-1));
  new_path_flag(i)   = u(1+4*(i-1)+2*P);
  traj(:,i)          = u(2+4*(i-1)+2*P:4+4*(i-1)+2*P);
end
num_threats          = u(1+6*P);
threats              = reshape(u(2+6*P:1+2*num_threats+6*P), ...
             num_threats,2);

for i=1:P,

  switch state(i),

   case 1, % check for new path request
    if new_path_flag(i)==1,
      state(i) = 2;
    elseif new_path_flag(i)==2,
      state(i) = 4;
    else
      state(i) = 1;
    end

   case 2, % simple target request
    new_target_flag(i) = 1;
    state(i) = 3;

   case 3, % make sure target changes
     if norm(target(:,i)-target_stored(:,i))~=0,
       state(i) = 4;
       new_target_flag(i)=0;
     else
       state(i) = 3;
     end

   case 4, %We have a new path - send to trajGen
    vel = (uavdata(i).v_max + uavdata(i).v_min)/2; % pick velocity inbetween min and max
    [vp, vel] = generateVPpath(threats, traj(:,i), target(:,i), vel);
    path_len = size(vp,2);
    path(:,i) = [path_len;...
             vel;...
             reshape([vp(2:3,:); uavdata(i).ic(5)*ones(1,path_len)],...
                 3*path_len, 1);...
```

```
            zeros(3*(M-path_len),1)];
        state(i) = 5;

      case 5, % wait for acknowledgment from trajGen
        if new_path_flag(i)==0,
          state(i)=1;
        else
          state(i)=5;
        end

      otherwise
        disp('Unknown state')
    end
  end

  xup = [...
        num_threats;...
        state;...
        reshape(path,(2+3*M)*P,1);...
        reshape(target(:,i),2*P,1);...
                new_target_flag;...
    ];

%end mdlUpdate
%
%=============================================================================
% mdlOutputs
% Return Return the output vector for the S-function
%=============================================================================
%
function y = mdlOutputs(t,x,u,uavdata,env,M,P)

  %input('In wpp output');
  for i=1:P,
    path(:,i)          = x(1+(2+3*M)*(i-1)+P+1:(2+3*M)*i+P+1);
    new_target_flag(i) = x(i+1+P+(2+3*M)*P+2*P);
  end


  % output the paths for each UAV
  y = [new_target_flag; reshape(path,(2+3*M)*P,1)];

%end mdlUpdate
```

10.2 Supporting C files

```
/*----------------------------------------*/ /* Name: rungkut.c
*/ /* Created: 8/1/2002                    */ /*
*/ /* Does a 4th order Runge-Kutta         */ /* approximation
to the continuous part of */ /* g_uav - i.e. x, y, and psi
*/ /*----------------------------------------*/

#include "trajGen.h"

extern uavInfo g_uav[MAXPLANES];        /**< global variable
holding all uav info - declared in trajGen.c */ extern uavInfo
g_uav_prev[MAXPLANES];   /**< global variable holding the previous
uav info for g_uav  - declared in trajGen.c */

/** Self contained function to run 5th order Runge-Kutta-Fehlberg
*/ void rungkutfehl(double, double, int, int, double, double);

/** global function pointer used to point to yprime to be used in
Runge-Kutta algos */ double (*g_yprimeptr) (double, double);

/** function that specifies yprime and is pointed to by
g_yprimeptr */ double yprime(double, double);

/* int main(){

  int N = 10;
  int i = 0;
  double t = 0;
  double w = 1;
  g_yprimeptr = yprime;
  for(;i<N;i++){

    printf("%-1.1lf\t%-1.10lf\n",t,w);
    w = rungkut4(0.1,t,w);
    t += .1;
  }
  printf("%-1.1lf\t%-1.10lf\n",t,w);

  rungkutfehl(0.1,0.1,0,1,1,pow(10.0,-5.0));
} */

void uavCopy(int id, uavInfo* K)  // copies g_uav_prev to K. This
so iterations in the algorithm {                          // don't
mess up g_uav_prev. We use the previous g_uav information
  int i;                  // to match Matlab sequence and to avoid making a dependency on
```

```
                        // order of calls to rungkut and stateUpdate
  K->x = g_uav_prev[id].x;
  K->y = g_uav_prev[id].y;
  K->z = g_uav_prev[id].z;
  K->psi = g_uav_prev[id].psi;
  K->Vel = g_uav_prev[id].Vel;
  K->turnFlag = g_uav_prev[id].turnFlag;
  K->state = g_uav_prev[id].state;
  K->newSigmaFlag = g_uav_prev[id].newSigmaFlag;

  for(i=0;i<N;i++)
  {
    K->currentSigma[i].x = g_uav_prev[id].currentSigma[i].x;
    K->currentSigma[i].y = g_uav_prev[id].currentSigma[i].y;
    K->currentSigma[i].z = g_uav_prev[id].currentSigma[i].z;
  }

  K->startNormal.x = g_uav_prev[id].startNormal.x;
  K->startNormal.y = g_uav_prev[id].startNormal.y;
  K->startNormal.z = g_uav_prev[id].startNormal.z;
  K->endNormal.x = g_uav_prev[id].endNormal.x;
  K->endNormal.y = g_uav_prev[id].endNormal.y;
  K->endNormal.z = g_uav_prev[id].endNormal.z;
}


// does 4th order Runge-Kutta approximation
// CHANGE: the z value (height) is approximated the same
// way as x, y, and psi, but 'trajGenDerivative' currently
// does not return the derivative of z, so the approximation
// is not reliable with respect to z
void rungkutTrajGen(int id, double vel, planeInfo* Param, double
time_step){

  uavInfo K1;
  uavInfo K2;
  uavInfo K3;
  uavInfo K4;

  uavCopy(id, &K1);
  uavCopy(id, &K2);
  uavCopy(id, &K3);
  uavCopy(id, &K4);
  // now K1-K4 all hold the same information as g_uav_prev

  trajGenDerivative(&K1, vel, Param);  // find 1st approximation
```

```
    K1.x *= time_step;
    K1.y *= time_step;
    K1.z *= time_step;
    K1.psi *= time_step;

    // use K1 to modify K2 for setup to 2nd approximation
    K2.x += K1.x/2;
    K2.y += K1.y/2;
    K2.z += K1.z/2;
    K2.psi += K1.psi/2;
    trajGenDerivative(&K2, vel, Param);  // find 2nd approximation
    K2.x *= time_step;
    K2.y *= time_step;
    K2.z *= time_step;
    K2.psi *= time_step;

    // use K2 to modify K3 for setup to 3rd approximation
    K3.x += K2.x/2;
    K3.y += K2.y/2;
    K3.z += K2.z/2;
    K3.psi += K2.psi/2;
    trajGenDerivative(&K3, vel, Param);  // find 3rd approximation
    K3.x *= time_step;
    K3.y *= time_step;
    K3.z *= time_step;
    K3.psi *= time_step;

    // use K3 to modify K4 for setup to 4th approximation
    K4.x += K3.x;
    K4.y += K3.y;
    K4.z += K3.z;
    K4.psi += K3.psi;
    trajGenDerivative(&K4, vel, Param);  // find 4th approximation
    K4.x *= time_step;
    K4.y *= time_step;
    K4.z *= time_step;
    K4.psi *= time_step;

    // all approximations are done - use to find the new g_uav
    g_uav[id].x += (K1.x + 2*K2.x + 2*K3.x + K4.x)/6.0;
    g_uav[id].y += (K1.y + 2*K2.y + 2*K3.y + K4.y)/6.0;
    //g_uav.z += (K1.z + 2*K2.z + 2*K3.z + K4.z)/6.0;
    g_uav[id].z = 1;
    g_uav[id].psi += (K1.psi + 2*K2.psi + 2*K3.psi + K4.psi)/6.0;
}
```

```
// Approximation the finds best time step - however, since our time step
// is fixed, this cannot be used
void rungkutfehl(double hmax, double hmin, int a, int b, double
alpha, double tol){

  int flag = 1;
  double R;
  double delta;
  double h = hmax;
  double w = alpha;
  double t = a;

  printf("%1.8lf\t%1.8lf\ttol = %lf\n",t,w,tol);

  while(flag){

    double K1 = h*g_yprimeptr(w,t);
    double K2 = h*g_yprimeptr(w + K1/4,t+h/4);
    double K3 = h*g_yprimeptr(w + 9*K2/32 + 3*K1/32,t+3*h/8);
    double K4 = h*g_yprimeptr(w + 7296*K3/2197 + -7200*K2/2197 + 1932*K1/2197,t + 12*h/13);
    double K5 = h*g_yprimeptr(w + -845*K4/4104 + 3680*K3/513 + -8*K2 + 439*K1/216,t + h);
    double K6 = h*g_yprimeptr(w + -11*K5/40 + 1859*K4/4104 + -3544*K3/2565 + 2*K2 + -8*K1/2

    R = fabs(K1/360 - 128*K3/4275 - 2197*K4/75240 + K5/50 + 2*K6/55)/h;

    delta = .84*pow((tol/R),0.25);

    if(R<=tol){
      t = t + h;
      w = w + 25*K1/216 + 1408*K3/2565 + 2197*K4/4104 - K5/5;
      printf("%1.7lf\t%1.7lf\t%1.7lf\n",t,w,h);
    }

    if(delta<=.1){
      h = .1*h;
    }
    else if(delta>=4){
      h = 4*h;
    }
    else{
      h = delta*h;
    }

    if(h > hmax)
      h = hmax;
```

```
    if(t>= b)
      flag = 0;
    else if(t + h > b)
      h = b-t;
    else if(h < hmin){
      flag = 0;
      printf("minimum h exceeded******\n");
      return;
    }
  }
}


double yprime(double y, double t){

  return ( y );
} /*---------------------------------------*/ /* Name: trajGen.c
*/ /* Created: 8/5/2002                     */
/*---------------------------------------*/ /** This file
contains an implementation for the S-function
 *  used in the trajectory generation program
 */

#include "trajGen.h"
//#define UTURN_DEBUG 1
//#define DEBUG 1

uavInfo g_uav[MAXPLANES];          /**< global variable holding all
uav info */ uavInfo g_uav_prev[MAXPLANES];    /**< global variable
holding the previous uav info for g_uav */ int PLANE_PTR = 0;

void uav_prev_Copy(int id);        /**< copies g_uav to g_uav_prev
to unlink stateUpdate from rungkut */ int
isLeftOfLine(vect3*,vect3*,double,double);

// performs rungkut and state update for trajectory generation
void trajGenUpdate(int id, double vel, vect3 *sigma, planeInfo*
Param, double time_step) {
  /** call stateUpdate to update the state machine */
  g_uav[id].Vel = vel;
  stateUpdate(&g_uav[id], vel, sigma, Param);

  /** calculate the answer to the derivates - sets g_uav */
  rungkutTrajGen(id, vel,Param,time_step);

  /** copy g_uav to g_uav_prev */
```

```
  uav_prev_Copy(id);

  return;
}

// initializes all of the fields in g_uav
int trajGenInitialize(planeInfo* Param, double* wps0) {
  if(PLANE_PTR == MAXPLANES)
    return -1;

  g_uav[PLANE_PTR].x =    Param->ic.x0;
  g_uav[PLANE_PTR].y =    Param->ic.y0;
  g_uav[PLANE_PTR].z =    Param->ic.z0;
  g_uav[PLANE_PTR].psi =  Param->ic.psi0;
  g_uav[PLANE_PTR].Vel =  Param->ic.v0;

  g_uav[PLANE_PTR].turnFlag = 0;
  g_uav[PLANE_PTR].state = 0;

  g_uav[PLANE_PTR].currentSigma[0].x = wps0[0];
  g_uav[PLANE_PTR].currentSigma[0].y = wps0[1];
  g_uav[PLANE_PTR].currentSigma[0].z = wps0[2];
  g_uav[PLANE_PTR].currentSigma[1].x = wps0[3];
  g_uav[PLANE_PTR].currentSigma[1].y = wps0[4];
  g_uav[PLANE_PTR].currentSigma[1].z = wps0[5];
  g_uav[PLANE_PTR].currentSigma[2].x = wps0[6];
  g_uav[PLANE_PTR].currentSigma[2].y = wps0[7];
  g_uav[PLANE_PTR].currentSigma[2].z = wps0[8];

  g_uav[PLANE_PTR].newSigmaFlag = 0;
  g_uav[PLANE_PTR].startNormal.x = 0;
  g_uav[PLANE_PTR].startNormal.y = 0;
  g_uav[PLANE_PTR].startNormal.z = 0;
  g_uav[PLANE_PTR].endNormal.x = 0;
  g_uav[PLANE_PTR].endNormal.y = 0;
  g_uav[PLANE_PTR].endNormal.z = 0;

  uav_prev_Copy(PLANE_PTR); // make sure g_uav_prev contains the initial values

  PLANE_PTR++;
  return (PLANE_PTR - 1);
}

// performs state update for trajectory generation
void stateUpdate(uavInfo* uav, double vel, vect3 *wpN, planeInfo*
Param) {
```

```
  vect3 cl,cr,q1,q2,qTmp,c,v,n;
  double vectorNorm, a,a2, b,b2, d, psidot, normal_ang,beta,kappa = Param->kappa, eps_d = 0
  int hor,hor2,turn,turn2,clin,s,next_state,p,acc;
  double radius = vel/Param->psidot_max;          // radius of the 2 side circles of the pl
  double df = 2*radius;                            // distance that is considered far from

#ifdef UTURN_DEBUG
  printf("guav state is %d\n",uav->state);
#endif #ifdef DEBUG
  printf("TRAJGEN: --> START stateUpdate\n");
#endif

  if( (uav->newSigmaFlag == 0) &&
      (normN(wpN, uav->currentSigma) > 0.001) )   // if the input waypoints differ signific
  {                                                // from the current sigma, then we are r
    // We have a new set of waypoints given the manager
    sigmaCopy(wpN, uav->currentSigma);            // copy the input waypoints to current si
    uav->turnFlag = 0;
    uav->state = 0;

#ifdef DEBUG
    printf("TRAJGEN:     New set of waypoints\n");
    printf("TRAJGEN: --> END stateUpdate\n");
#endif

    return;
  }

  // find center of circle on left of uav
  cl.x = uav->x + radius*sin(uav->psi);      // x coordinate of center of circle on left
  cl.y = uav->y - radius*cos(uav->psi);      // y coordinate of center of circle on left
  cl.z = uav->z;                                  // z coordinate of center of circle on le

  // find center of circle on right of uav
  cr.x = uav->x - radius*sin(uav->psi);      // x coordinate of center of circle on right
  cr.y = uav->y + radius*cos(uav->psi);      // y coordinate of center of circle on right
  cr.z = uav->z;                                  // z coordinate of center of circle on ri

  // q1 and q2 are unit vectors along the current and next path edges respectively
  q1.x = uav->currentSigma[1].x - uav->currentSigma[0].x;
  q1.y = uav->currentSigma[1].y - uav->currentSigma[0].y;
  makeUnitVector(&q1);

  q2.x = uav->currentSigma[2].x - uav->currentSigma[1].x;
  q2.y = uav->currentSigma[2].y - uav->currentSigma[1].y;
  makeUnitVector(&q2);
```

```
  turn = turndir(q1,q2);  // find out which way the uav should turn -> 0, no turn
                          //                                          1, counter-clockwise
                          //                                         -1, clockwise (right-
                          //                                          2, u-turn
  if(turn != 2 && fabs(acos(q1.x*q2.x + q1.y*q2.y)) <= PI/10){
#ifdef DEBUG
    printf("TRAJGEN:     Shallow turn, setting turn to 0\n");
#endif
    turn = 0;
  }
#ifdef DEBUG
  printf("Update turn is %d\n",turn);
#endif

  // find the line equation y=ax+b for the line from sigma1 to sigma2
  hor = getlinevals(uav->currentSigma[1],uav->currentSigma[2],&a,&b);
  d = dist2path(uav->currentSigma[0],q1,uav->x,uav->y);  // how far off q1 the uav is

  acc = 12;       /**< accuracy that we know the equivalent distance kappa to is 2^(-acc) *
  next_state = 1; /**< default next state */

  switch(uav->turnFlag)   // where we are at in a turn: 0, track a line
    {                     //                          1, finish the last part of kappa t
                          //                          2, track a line until a new waypoi
                          //                          3,
                          //                          4,
                          //                          5,
                          //                          6,
                          //                          7,
                          //                          8, finish the last part of kappa t

    case 0:   // track a line
#ifdef DEBUG
      printf("TRAJGEN:     STATE 0\n");
#endif
       switch(Param->turn_param)  // how we execute turns: 0, QUESTION: what is this?
    {                        //                       1, parameterized turn in trajectory
    case 0:
#ifdef DEBUG
      printf("TRAJGEN:     Turn param is set to 0\n");
#endif
      // figure out the unit vector from the begining waypoint to the uav
      qTmp.x = uav->x - uav->currentSigma[0].x;
      qTmp.y = uav->y - uav->currentSigma[0].y;
      makeUnitVector(&qTmp);
```

```
      turn2 = turndir(qTmp,q1);  // find which direction we need to turn to get back on the
      // find the line equation 'y = a2x + b2' for the line from sigma0 to sigma1
      hor2 = getlinevals(uav->currentSigma[0],uav->currentSigma[1],&a2,&b2);

      // check to see if we are too far away from path
      d = dist2path(uav->currentSigma[0],q1,uav->x,uav->y);
      if( d >= df )
        {
#ifdef DEBUG
          printf("TRAJGEN:     Uav is far off the path with turn param set to 0\n");
#endif
          if ( (clintersect(hor2,a2,b2,cl.x,cl.y,radius,uav->x,uav->y) && (turn2 < 0)) ||
           (clintersect(hor2,a2,b2,cr.x,cr.y,radius,uav->x,uav->y) && (turn2 > 0)) )
        uav->turnFlag = 5;
          else
        {
          uav->turnFlag = 3;
          // make sure that our normal is pointing towards the path
          if( sign((uav->x-uav->currentSigma[0].x)*q1.y - (uav->y-uav->currentSigma[0].y)*q
            normal_ang = atan2(q1.x,-q1.y);
          else
            normal_ang = atan2(-q1.x,q1.y);
          next_state = sign(sin(normal_ang - uav->psi));
        }
           }
      if ( (clintersect(hor,a,b,cl.x,cl.y,radius,uav->x,uav->y) && (turn < 0)) ||
           (clintersect(hor,a,b,cr.x,cr.y,radius,uav->x,uav->y) && (turn > 0)) )
        {
#ifdef DEBUG
          printf("TRAJGEN:     Switching to turning with turn param set to 0\n");
#endif
          uav->turnFlag = 1;
          next_state = sign(cos(uav->psi)*q2.y - sin(uav->psi)*q2.x);
        }
      break;

    case 1:
#ifdef DEBUG
      printf("TRAJEGN:     Turn param set to 1, i.e. parameterized turns\n");
#endif
      qTmp.x = uav->x - uav->currentSigma[0].x;
      qTmp.y = uav->y - uav->currentSigma[0].y;
      makeUnitVector(&qTmp);

      turn2 = turndir(qTmp,q1);
```

```
      hor2 = getlinevals(uav->currentSigma[0],uav->currentSigma[1],&a2,&b2);

      /** check to see if we are too far away from path */
      d = dist2path(uav->currentSigma[0],q1,uav->x,uav->y);
      if( d >= df )
        {
#ifdef DEBUG
          printf("TRAJGEN:     Uav far off the path with turn param set to 1\n");
#endif
          if ( (clintersect(hor2,a2,b2,cl.x,cl.y,radius,uav->x,uav->y) && (turn2 < 0)) ||
           (clintersect(hor2,a2,b2,cr.x,cr.y,radius,uav->x,uav->y) && (turn2 > 0)) )
        uav->turnFlag = 5;
          else
        {
          uav->turnFlag = 3;
          // make sure that our normal is pointing towards the path
          if( sign((uav->x-uav->currentSigma[0].x)*q1.y - (uav->y-uav->currentSigma[0].y)*q
            normal_ang = atan2(q1.x,-q1.y);
          else
            normal_ang = atan2(-q1.x,q1.y);
          next_state = sign(sin(normal_ang - uav->psi));
        }
        }

      //CHANGE: there has to be a better way to make sure that the uav angle
      //        is close to match the line it's on
      beta = acos(qTmp.x*q1.x+qTmp.y*q1.y);
      if(fabs(beta) > 10*PI/180 ){
#ifdef DEBUG
        printf("TRAJGEN:     Heading far off\n");
#endif
        break;
      }

      beta = fabs(PI - acos(q2.x*q1.x + q2.y*q1.y));
      if( Param->equivdist )
        {
          kappa = GetKappa(radius,beta,acc);
            }
      if(turn == 2)
      {
#ifdef DEBUG
        printf("TRAJGEN:     Calculating U-turn\n");
#endif
        getUturncenter(uav->currentSigma[0], uav->currentSigma[1],radius,&c);
      }
```

```
      else
      {
#ifdef DEBUG
        printf("TRAJGEN:     Calculating %d turn\n", turn);
#endif
        getturncenter(uav->currentSigma[1],q1,q2,radius,kappa,beta,&c);
      }
      if ( (ccintersect(uav,c,radius,cl,radius) && (turn > 0)) ||
           (ccintersect(uav,c,radius,cr,radius) && (turn < 0)) )
        {
#ifdef DEBUG
          printf("TRAJGEN:     Starting %d turn\n", turn);
#endif
          uav->turnFlag = 6;
          // printf("Kappa: %lf\n",kappa);
          if( turn > 0 )
        {
          v.x = cl.x - c.x;
          v.y = cl.y - c.y;
          n.x = -v.y;
          n.y = v.x;
        }
          else if( turn < 0 )
        {
          v.x = cr.x - c.x;
          v.y = cr.y - c.y;
          n.x = -v.y;
          n.y = v.x;
        }
          else
        {
          n.x = 0;
          n.y = 0;
        }
          // QUESTION: what is happening here?
          next_state = sign(cos(uav->psi)*n.y - sin(uav->psi)*n.x);
        }
      if(turn == 0)
      {
#ifdef DEBUG
        printf("TRAJGEN:     Turn is equal to 0\n");
#endif
        // go a dist radius/4 along sigma1-sigma0 toward sigma0
        // if that point is not on the same side of simga1-sigma2 as the uav is
        // then go to state 2
        q1.x = uav->currentSigma[1].x;
```

```
        q1.y = uav->currentSigma[1].y;
        q2.x = uav->currentSigma[2].x;
        q2.y = uav->currentSigma[2].y;
        qTmp.x = uav->x;
        qTmp.y = uav->y;

        s = getSideOfLine(&q1,&q2,&uav->currentSigma[0]);
        if(!s)
        {
#ifdef DEBUG
          printf("TRAJGEN:     The line from wp1 to wp2 is the same as wp0 to wp1\n");
#endif
          // its the same line
          // check x = b
          if(hor)
          {
        s = sign(uav->currentSigma[0].y-uav->currentSigma[1].y);
        if(s != sign(uav->y - uav->currentSigma[1].y))
        {
#ifdef DEBUG
          printf("TRAJGEN:     Switching to next line because next line is the same line");
#endif
          uav->turnFlag = 2;
        }
          }
          else
          {
        // use x values to determine
        s = sign(uav->currentSigma[0].x-uav->currentSigma[1].x);
        if(s != sign(uav->x - uav->currentSigma[1].x))
        {
#ifdef DEBUG
          printf("TRAJGEN:     Switching to next line because next line is the same line");
#endif
          uav->turnFlag = 2;
        }
          }
        }
        else
        {
          offsetLine(&q1,&q2,radius/4.0,s);
          if(getSideOfLine(&q1,&q2,&qTmp) != s)
          {
#ifdef DEBUG
        printf("TRAJGEN:     Switching to next line because turn too shallow\n");
#endif
```

```
        uav->turnFlag = 2;
          }
        }
      }
      break;
    default:
      printf("Unhandled turn_param = %d\n",Param->turn_param);
        }
      break;

    case 1:
#ifdef DEBUG
      printf("TRAJGEN:     STATE 1\tAre we close enough to the next path?\n");
#endif
      // This condition determines when we are close enough to the next path
      //  to stop turning
      next_state = sign(cos(uav->psi)*q2.y - sin(uav->psi)*q2.x);
      if( (uav->state != next_state) || (next_state == 0) )
    uav->turnFlag = 2;
      break;

    case 2:
#ifdef DEBUG
      printf("TRAJGEN:     STATE 2\tRequesting new waypoint\n");
#endif
      if( uav->newSigmaFlag == 0 )
    {
      uav->newSigmaFlag = 1;
    }
      else if( normN(wpN,uav->currentSigma) > 0.001)    /**< this next elseif happens if we
    {
      uav->newSigmaFlag = 0;
      uav->turnFlag = 0;
      sigmaCopy(wpN,uav->currentSigma);
    }
      break;

    case 3:
#ifdef DEBUG
      printf("TRAJGEN:     STATE 3\tStarting non-parameterized turn\n");
#endif
      qTmp.x = cos(uav->psi);
      qTmp.y = sin(uav->psi);
      makeUnitVector(&qTmp);

      turn2 = turndir(qTmp,q1);
```

```
      hor2 = getlinevals(uav->currentSigma[0],uav->currentSigma[1],&a2,&b2);

      next_state = sign((uav->x-uav->currentSigma[0].x)*q1.y - (uav->y-uav->currentSigma[0]

      if ( (clintersect(hor2,a2,b2,cl.x,cl.y,radius,uav->x,uav->y) && (turn2 < 0)) ||
       (clintersect(hor2,a2,b2,cr.x,cr.y,radius,uav->x,uav->y) && (turn2 > 0)) )
    uav->turnFlag = 5;
      else
    {
      s = sign((uav->x-uav->currentSigma[0].x)*q1.y - (uav->y-uav->currentSigma[0].y)*q1.x)
      // make sure that our normal is pointing towards the path
      if( s >= 0 )
        {
          normal_ang = atan2(q1.x,-q1.y);
          uav->endNormal.x = -q1.y;
          uav->endNormal.y = q1.x;
        }
      else
        {
          normal_ang = atan2(-q1.x,q1.y);
          uav->endNormal.x = q1.y;
          uav->endNormal.y = -q1.x;
        }
      p = sign(sin(normal_ang - uav->psi));
      // determine if turn is finished
      if( (p != uav->state) || (p == 0) )
        {
          uav->turnFlag = 4;
          uav->startNormal.x = uav->x;
          uav->startNormal.y = uav->y;
        }
        }
      break;

    case 4:
#ifdef DEBUG
      printf("TRAJGEN:     STATE 4\tMiddle of non parameterized turn\n");
#endif
      qTmp.x = uav->endNormal.x;
      qTmp.y = uav->endNormal.y;
      makeUnitVector(&qTmp);
      turn2 = turndir(qTmp,q1);
      hor2 = getlinevals(uav->currentSigma[0],uav->currentSigma[1],&a2,&b2);
      next_state = sign(cos(uav->psi)*q1.y - sin(uav->psi)*q1.x);
      if ( (clintersect(hor2,a2,b2,cl.x,cl.y,radius,uav->x,uav->y) && (turn2 < 0)) ||
       (clintersect(hor2,a2,b2,cr.x,cr.y,radius,uav->x,uav->y) && (turn2 > 0)) )
```

```
    uav->turnFlag = 5;
      break;

    case 5:
#ifdef DEBUG
      printf("TRAJGEN:     STATE 5\tLast part of non parameterized turn\n");
#endif
      next_state = sign(cos(uav->psi)*q1.y - sin(uav->psi)*q1.x);
      if( (uav->state != next_state) || (next_state == 0) )
    uav->turnFlag = 0;
      break;

    case 6:
#ifdef DEBUG
      printf("TRAJGEN:     STATE 6\tStarting Kappa turn\n");
#endif
      beta = fabs(PI - acos(q2.x*q1.x + q2.y*q1.y));
      if( Param->equivdist )
    kappa = GetKappa(radius,beta,acc);

      if(turn == 2) getUturncenter(uav->currentSigma[0], uav->currentSigma[1], radius,&c);
      else
    getturncenter(uav->currentSigma[1],q1,q2,radius,kappa,beta,&c);
      if( turn > 0 )
    {
      v.x = cl.x - c.x;
      v.y = cl.y - c.y;
      n.x = -v.y;
      n.y = v.x;
    }
      else if( turn < 0 )
    {
      v.x = cr.x - c.x;
      v.y = cr.y - c.y;
      n.x = -v.y;
      n.y = v.x;
    }
      else
    {
      n.x = 0;
      n.y = 0;
    }

      next_state = sign(cos(uav->psi)*n.y - sin(uav->psi)*n.x);
      if( (uav->state != next_state) || (next_state == 0) )
    {
```

```
      uav->turnFlag = 7;
      next_state = 0;
    }
      break;

   case 7:
#ifdef DEBUG
      printf("TRAJGEN:     STATE 7\tMiddle of Kappa turn\n");
#endif #ifdef UTURN_DEBUG
      printf("Current turn is: %d\n",turn);
      if(turn == 2){
    printf("(%.3lf,%.3lf)\t(%.3lf,%.3lf)\t(%.3lf,%.3lf)\n",
           uav->currentSigma[0].x,
           uav->currentSigma[0].y,
           uav->currentSigma[1].x,
           uav->currentSigma[1].y,
           uav->currentSigma[2].x,
           uav->currentSigma[2].y);
    printf("State 7\n");
      }
#endif
      if ( (clintersect(hor,a,b,cl.x,cl.y,radius,uav->x,uav->y) && (turn == 1 || (turn == 2
       (clintersect(hor,a,b,cr.x,cr.y,radius,uav->x,uav->y) && (turn == -1)) ||
       (isLeftOfLine(&uav->currentSigma[0],&uav->currentSigma[1],uav->x,uav->y) && (turn ==
      {
#ifdef UTURN_DEBUG
    if(turn == 2){
      printf("There was a clintersect with %d, %.2lf, %.2lf, (%.2lf,%.2lf)\tuav -> (%.2lf,%
    }
#endif
    if( (uav->state == 0 && turn != 2) || (uav->state == 1)){
#ifdef UTURN_DEBUG
      if(turn == 2){
        printf("Setting uav->state to 1\n");
      }
#endif
      next_state = 1;
    }
    else if(uav->state == 2)
    {
#ifdef UTURN_DEBUG
      if(turn == 2){
        printf("Switching to state 8 now\n");
      }
#endif
      uav->turnFlag = 8;
```

```
      next_state = sign(cos(uav->psi)*q2.y - sin(uav->psi)*q2.x);
    }
      }
      else if(uav->state == 1){
#ifdef UTURN_DEBUG
    if(turn == 2){
      printf("Setting uav->state to 2\n");
    }
#endif
    next_state = 2;
      }
      else
    next_state = uav->state;
      break;

    case 8:
#ifdef DEBUG
      printf("TRAJGEN:     STATE 8\tFinishing Kappa turn\n");
#endif
      // This condition determines when we are close enough to the next path
      // to stop turning
      next_state = sign(cos(uav->psi)*q2.y - sin(uav->psi)*q2.x);
      if( (uav->state != next_state) || (next_state == 0) )
    uav->turnFlag = 2;
      break;
    }
#ifdef DEBUG
  printf("TRAJGEN: --> END stateUpdate\n");
#endif
  uav->state = next_state;
}

int trajGenOutput(int id, double vel, double* statevars, double
timestep, planeInfo* Param) {
  int i;
  static double last_psi[MAXPLANES];
  static double last_vel[MAXPLANES];
  static int init_derivative[MAXPLANES];
  static int allinit = 1;
  uavInfo K1;

  if(allinit){
    for(i=0; i<MAXPLANES; i++)
      init_derivative[i] = 1;
    allinit = 0;
  }
```

```
  statevars[0] = g_uav[id].x;
  statevars[1] = g_uav[id].y;
  statevars[2] = g_uav[id].psi;
  statevars[3] = vel;

  uavCopyCurrent(id, &K1);
  trajGenDerivative(&K1, vel, Param);

  statevars[4] = K1.x;
  statevars[5] = K1.y;
  statevars[6] = K1.psi;
  statevars[7] = K1.Vel;

  if(init_derivative[id]){
    last_psi[id] = K1.psi;
    last_vel[id] = K1.Vel;
    init_derivative[id] = 0;
  }

  statevars[8]  = K1.Vel*cos(g_uav[id].psi) - vel*(K1.psi)*cos(g_uav[id].psi);
  statevars[9]  = K1.Vel*sin(g_uav[id].psi) + vel*(K1.psi)*sin(g_uav[id].psi);
  statevars[10] = (K1.psi - last_psi[id])/timestep;
  statevars[11] = (K1.Vel - last_vel[id])/timestep;

  last_psi[id] = K1.psi;
  last_vel[id] = K1.Vel;

  return g_uav[id].newSigmaFlag;
}

void trajGenDerivative(uavInfo* uav, double vel, planeInfo* Param)
{
  vect3 cl,cr,q1,q2,qTmp;
  double vectorNorm, a, b, d, psidot, eps_d = 0.0001;
  int hor,turn,clin,s;
  double radius = vel/Param->psidot_max; /**< radius of the 2 side circles of the plane */
  double df = 2*radius;                  /**< distance that is considered far from

  cl.x = uav->x + radius*sin(uav->psi);  /**< x coordinate of center of circle on l
  cl.y = uav->y - radius*cos(uav->psi);  /**< y coordinate of center of circle on l
  cl.z = uav->z;

  cr.x = uav->x - radius*sin(uav->psi);  /**< x coordinate of center of circle on r
  cr.y = uav->y + radius*cos(uav->psi);  /**< y coordinate of center of circle on r
  cr.z = uav->z;
```

```
  /** q1 and q2 are unit vectors along the current and next path edges respectively */
  q1.x = uav->currentSigma[1].x - uav->currentSigma[0].x;
  q1.y = uav->currentSigma[1].y - uav->currentSigma[0].y;
  q1.z = uav->z;
  makeUnitVector(&q1);

  q2.x = uav->currentSigma[2].x - uav->currentSigma[1].x;
  q2.y = uav->currentSigma[2].y - uav->currentSigma[1].y;
  q2.z = uav->z;
  makeUnitVector(&q2);

  // compute distances
  hor = getlinevals(uav->currentSigma[1],uav->currentSigma[2],&a,&b);
  turn = turndir(q1,q2);
#ifdef DEBUG
  printf("Derivative turn is: %d\n",turn);
#endif
  d = dist2path(uav->currentSigma[0],q1,uav->x,uav->y);

  // turn_initiated
  switch (uav->turnFlag)
  {
    case 0:
      switch (Param->turn_param)
      {
    case 0:
      psidot = track(uav->currentSigma[0],q1,vel,uav->x,uav->y,uav->psi,eps_d,Param->psidot

      if(turn > 0 && clintersect(hor,a,b,cr.x,cr.y,radius,uav->x,uav->y))
        psidot = Param->psidot_max;
          else if (turn < 0 && clintersect(hor,a,b,cl.x,cl.y,radius,uav->x,uav->y))
        psidot = -Param->psidot_max;

      break;
        case 1:
      psidot = track(uav->currentSigma[0],q1,vel,uav->x,uav->y,uav->psi,eps_d,Param->psidot
      break;
        default:
          printf("Unhandled turn_param = %d",Param->turn_param);
      }
      break;

    case 1:
      if (dist2path(uav->currentSigma[1],q2,uav->x,uav->y) < eps_d)
    psidot = 0;
```

```
  else if (turn > 0)   /**< keep turning */
psidot = Param->psidot_max;
  else if (turn < 0)
psidot = -Param->psidot_max;
  else /**< otherwise our next path segment is straight */
psidot = 0;
  break;

/**a turn is not finished until we have a new waypoint */
case 2:
  psidot = track(uav->currentSigma[0],q1,vel,uav->x,uav->y,uav->psi,eps_d,Param->psidot
  break;

case 3:
  /** s gives the direction of the turn - in the direction of q1 */
  s = sign((uav->x - uav->currentSigma[0].x)*q1.y - (uav->y - uav->currentSigma[0].y)*q
  qTmp.x = cos(uav->psi);
  qTmp.y = sin(uav->psi);
  qTmp.z = uav->z;
  makeUnitVector(&qTmp);
  turn = turndir(qTmp,q1);
  hor = getlinevals(uav->currentSigma[0],uav->currentSigma[1],&a,&b);

  if (turn > 0)                    /**< turn to path if close enough */
  {
if( clintersect(hor,a,b,cr.x,cr.y,radius,uav->x,uav->y) )
  psidot = Param->psidot_max;
else                             /**< otherwise turn to normal path */
  psidot = s*Param->psidot_max;
  }
  else if (turn < 0)               /**< turn to path if close enough */
  {
if( clintersect(hor,a,b,cl.x,cl.y,radius,uav->x,uav->y) )
  psidot = -Param->psidot_max;
else                             /**< otherwise turn to normal path */
  psidot = s*Param->psidot_max;
  }
  else
psidot = track(uav->currentSigma[0],q1,vel,uav->x,uav->y,uav->psi,eps_d,Param->psidot_m
  break;

case 4:
  psidot = track(uav->startNormal,uav->endNormal,vel,uav->x,uav->y,uav->psi,eps_d,Param
  break;

case 5:
```

```
    qTmp.x = uav->endNormal.x;
    qTmp.y = uav->endNormal.y;
    makeUnitVector(&qTmp);
    turn = turndir(qTmp,q1);
    if(turn > 0)
  psidot = Param->psidot_max;
    else if(turn < 0)
  psidot = -Param->psidot_max;
    else
  psidot = 0;
    break;

  case 6:
    if(turn > 0)
  psidot = -Param->psidot_max;
    else if(turn < 0)
  psidot = Param->psidot_max;
    else
  psidot = 0;
    break;

  case 7:
    if(turn > 0)
  psidot = Param->psidot_max;
    else if(turn < 0)
  psidot = -Param->psidot_max;
    else
  psidot = 0;
    break;

  case 8:
    if(turn > 0)
  psidot = -Param->psidot_max;
    else if(turn < 0)
  psidot = Param->psidot_max;
    else
  psidot = 0;
    break;

  default:
    printf("Unhandled turn_initiated = %d",uav->turnFlag);
}

uav->x = vel*cos(uav->psi);
uav->y = vel*sin(uav->psi);
uav->psi = psidot;
```

```
  uav->Vel = 0; //10*(vel-uav->Vel);
}

// returns the values of the line a,b in y = ax + b, and hor is set to 0
// if the line is x = b, then hor is set to 1 and b is returned
int getlinevals(vect3 p1, vect3 p2, double* a, double* b) {
  double tol = 0.0001;
  /** The tolerance condition is necessary to prevent truncation errors
   * for nearly parallel lines */
  if(fabs(p1.x - p2.x) < tol)
  {
     *a = 0;
     *b = p1.x;
     return 1;
  }
  *a = (p1.y - p2.y) / (p1.x - p2.x);
  *b = -(*a)*p1.x + p1.y;
  return 0;
}

int turndir(vect3 q1, vect3 q2) {
  // if unit vectors are exactly opposite, just turn
  if( fabs(q1.x+q2.x) < 0.0001 && fabs(q1.y+q2.y) < 0.0001 )
    return 2;

  /** same as tmp = cross([q1 0],[q2 0]);
   *          turn = sign(tmp); */
  return sign( q1.x*q2.y - q1.y*q2.x );
}

int sign(double val) {
  if(val < 0) return -1;
  if(val > 0) return 1;
  return 0;
}

double norm(vect3 q) {
  return sqrt( q.x*q.x + q.y*q.y );
}

double dist2path(vect3 p, vect3 q, double X, double Y) {
  vect3 v;
  double proj_v;

  v.x = X - p.x;
  v.y = Y - p.y;
```

```
  v.z = p.z;

  proj_v = v.x*q.x + v.y*q.y;

  v.x -= proj_v*q.x;
  v.y -= proj_v*q.y;

  return norm(v);
}

double track(vect3 p, vect3 q, double v, double X, double Y,
double psi, double eps_d, double psidot_max, planeInfo* Param) {
  double alpha = 10.0;

  double psidot = alpha *
                  ( (2*psidot_max/PI)*atan(Param->k1*(q.y*(X - p.x) - q.x*(Y - p.y)))
           + Param->k2*v*(cos(psi)*q.y - sin(psi)*q.x) );

  if( psidot > psidot_max )
    psidot = psidot_max;
  else if( psidot < -psidot_max )
    psidot = -psidot_max;

  return psidot;
}

// returns true if specified circle intersects the next path segment
// and false if specified circle does not intersect the next path segment.
//---------------------------------------------------------------------
// hor indicates a horizontal path segment x = b, otherwise the path segment
// is represented as y = ax + b
// xc and yc are the x and y coordinates of the center of the circle to
// the left or right of the uav
// r is the radius of that circle
// X and Y are the current x and y position of the uav
int clintersect(int hor,double a, double b, double xc, double yc,
double r, double X, double Y) {
  double cival;
  if(hor) // a horizontal path segment
  {
    if( fabs(xc - b) <= r )
      return 1;
    else
      return 0;
  }
  else
```

```
  {
    // this messy expression is a discriminant
    cival = ( (2*a*(b-yc)-2*xc)*(2*a*(b-yc)-2*xc) - 4*(1 + a*a)*(xc*xc + (b-yc)*(b-yc) - r*
    if(cival >= 0)
      return 1;
    else
      return 0;
  }
  return 0;
}

/** this takes the norm of a N point waypoint path
 *  basically it's just to see if the path has changed */
double normN(vect3 s[], vect3 t[]) {
  int i = 0;
  double rval = 0.0;

  for(;i<N;i++)
  {
    rval += (s[i].x-t[i].x)*(s[i].x-t[i].x);
    rval += (s[i].y-t[i].y)*(s[i].y-t[i].y);
    /* rval += (s[i].z-t[i].z)*(s[i].z-t[i].z); */
  }

  return sqrt( rval );
}

void sigmaCopy(vect3 fromS[], vect3 toS[]) {
  int i = 0;

  for(;i<N;i++)
  {
    toS[i].x = fromS[i].x;
    toS[i].y = fromS[i].y;
    toS[i].z = fromS[i].z;
  }
}

// returns the angle between the path formed by the next
// three waypoints in the order w0, w1, w2
double getBeta(vect3* w0, vect3* w1, vect3* w2){
  /** q1 and q2 are unit vectors along the current and next path edges respectively */
  vect3 q1, q2;

  q1.x = w1->x - w0->x;
  q1.y = w1->y - w0->y;
```

```
  makeUnitVector(&q1);
  q2.x = w2->x - w1->x;
  q2.y = w2->y - w1->y;
  makeUnitVector(&q2);

  return fabs(PI - acos(q2.x*q1.x + q2.y*q1.y));

}

double GetKappa(double r, double beta, int n) {
  int i = 0;
  double L;
  double a = 0.0;
  double b = 1.0;
  for(;i<n;i++)
  {
    L = Lambda(r,(a + b)/2.0,beta);
    if(L == 0)
      return ( (a + b)/2.0 );
    else if(L > 0)
      b = (a + b)/2.0;
    else
      a = (a + b)/2.0;
  }
  return a;
}


double Lambda(double r, double k, double beta) {
  double sb = sin(beta/2.0);
  double sp = 1 + sb;
  double sm = 1 - sb;
  return ( r * ( sqrt(4.0 - (sp + k*sm)*(sp + k*sm) ) + (1.0 + k*(1.0/sb - 1.0))*cos(beta/2
          + (beta - PI)/2.0 - 2.0*acos((sp + k*sm)/2.0) ) );
}

void getturncenter(vect3 p, vect3 q1, vect3 q2, double radius,
double kappa, double beta, vect3* c) {
  vect3 v;
  double vectorNorm;

  v.x = q2.x - q1.x;
  v.y = q2.y - q1.y;

  vectorNorm = norm(v);
  if( vectorNorm == 0 )
```

```
  {
    v.x = 0;
    v.y = 0;
  }
  else
  {
    v.x = v.x/vectorNorm;
    v.y = v.y/vectorNorm;
  }

  c->x = p.x + radius*( 1.0 + kappa*(1/sin(beta/2.0) - 1) )*v.x;
  c->y = p.y + radius*( 1.0 + kappa*(1/sin(beta/2.0) - 1) )*v.y;
}

int ccintersect(uavInfo* uav, vect3 c1,double r1,vect3 c2,double
r2) {
  vect3 v, begining, ending, intercept;
  double vectorNorm, dist, x, y;
  int sx, sy;

  //if(c1.x == c2.x && c1.y == c2.y)
  if( (fabs(c1.x - c2.x) < 0.0001) && (fabs(c1.y - c2.y) < 0.0001) )
  {
    //if( r1 == r2 )

    if( fabs(r1-r2) < 0.0001 )
      return 1;
    return 0;
  }
  else
  {
    v.x = c1.x - c2.x;
    v.y = c1.y - c2.y;
    vectorNorm = norm(v);

    if( vectorNorm <= (r1 + r2) )
    {
      //if( r1 == r2 )

      if( fabs(r1-r2) < 0.0001 )
        return 1;
      else if( r1 < r2 )
      {
    v.x = 1.0 + r1/vectorNorm*v.x;
    v.y = 1.0 + r1/vectorNorm*v.y;
    if( norm(v) >= r2 )
```

```
      return 1;
    return 0;
      }
      else
      {
    v.x = 1.0 + r2/vectorNorm*v.x;
    v.y = 1.0 + r2/vectorNorm*v.y;
    if( norm(v) >= r1 )
      return 1;
    return 0;
      }
    }
    else  // we're possibly off the path
    {
      return ( offPathCCintersect(uav,c1,r1,c2) );
    }
  }
  return 0;
}

void uav_prev_Copy(int id) {
  int i = 0;

  g_uav_prev[id].x = g_uav[id].x;
  g_uav_prev[id].y = g_uav[id].y;
  g_uav_prev[id].z = g_uav[id].z;
  g_uav_prev[id].Vel = g_uav[id].Vel;
  g_uav_prev[id].psi = g_uav[id].psi;
  g_uav_prev[id].turnFlag = g_uav[id].turnFlag;
  g_uav_prev[id].state = g_uav[id].state;

  for(;i<N;i++)
  {
    g_uav_prev[id].currentSigma[i].x = g_uav[id].currentSigma[i].x;
    g_uav_prev[id].currentSigma[i].y = g_uav[id].currentSigma[i].y;
    g_uav_prev[id].currentSigma[i].z = g_uav[id].currentSigma[i].z;
  }

  g_uav_prev[id].newSigmaFlag = g_uav[id].newSigmaFlag;

  g_uav_prev[id].startNormal.x = g_uav[id].startNormal.x;
  g_uav_prev[id].startNormal.y = g_uav[id].startNormal.y;
  g_uav_prev[id].startNormal.z = g_uav[id].startNormal.z;

  g_uav_prev[id].endNormal.x = g_uav[id].endNormal.x;
  g_uav_prev[id].endNormal.y = g_uav[id].endNormal.y;
```

```
  g_uav_prev[id].endNormal.z = g_uav[id].endNormal.z;
}

void trajGenDestroy() {
  PLANE_PTR = 0;
}

// the old 'get distance to point' function from robot soccer -- Crankshaft rules!!
double getDistToLineAndIntersection(vect3* beg, vect3* end, vect3*
p, vect3* cept){

  double mdest, mpoint;
  double ydiff, xdiff;
  double ydiffdest, xdiffdest;
  double bdest, bpoint;

  ydiffdest = end->y - beg->y;
  xdiffdest = end->x - beg->x;

  if(!xdiffdest) mdest = 100000;
  else
    mdest = ydiffdest/xdiffdest;

  if(!mdest) mdest = 0.00001;
  mpoint = -1/mdest;

  bdest  = beg->y - mdest*beg->x;
  bpoint = p->y - mpoint*p->x;

  cept->x = (bpoint - bdest)/(mdest - mpoint);
  cept->y = (mdest*cept->x) + bdest;

  ydiff = p->y - cept->y;
  xdiff = p->x - cept->x;

  return sqrt(ydiff*ydiff+xdiff*xdiff);
}

void makeUnitVector(vect3* v) {
  double vectorNorm = norm(*v);
  if( vectorNorm == 0 )
  {
    v->x = 0;
    v->y = 0;
  }
  else
```

```
  {
    v->x = v->x/vectorNorm;
    v->y = v->y/vectorNorm;
  }
}

// for a vector with beginning point 'b' and ending point 'e' and a point 'p'
// this function returns -1 for 'p' on the right of the vector, 1 for 'p' on
// the left of the vector and 0 for 'p' on the vector
int getSideOfLine(vect3* b, vect3* e, vect3* p) {
  return sign( (e->x-b->x)*(p->y-b->y) - (e->y-b->y)*(p->x-b->x) );
}

// if the uav is flying too far off the path to intersect the circle
// then this function is called. It calculates the point where the
// circles should intersect, then it creates a line from that point
// perpendicular to the current path segment and checks to see if the
// circle has crossed the line. If it has it returns true
int offPathCCintersect(uavInfo* uav, vect3 c, double r, vect3
sideCircle) {
  vect3 begining,ending,intercept;
  double dist,gamma;
  int side;

  // set the line 'begining to end' to the current line
  // the plane is on
  begining.x = uav->currentSigma[0].x;
  begining.y = uav->currentSigma[0].y;
  ending.x   = uav->currentSigma[1].x;
  ending.y   = uav->currentSigma[1].y;
  begining.z = ending.z = 1;

  side = getSideOfLine(&begining,&ending,&uav->currentSigma[2]);
  if(!side) return 0;
  // offset to opposite side of sigma2
  offsetLine(&begining,&ending,r,-side);

  // get the distance to the offset line and the point of intersection
  dist = getDistToLineAndIntersection(&begining, &ending, &c, &intercept);
  // compute the distance needed to offset
  gamma = sqrt(4*r*r - dist*dist);

  side = getSideOfLine(&c,&intercept,&begining);
  // offset to the side of the begining point
  offsetLine(&c,&intercept,gamma,side);
```

```
  // the center of the side circle has crossed the line,
  // then we need to start turning
  if( getSideOfLine(&c,&intercept,&sideCircle) != side )
  {
#ifdef DEBUG
    printf("Returning 1 from offCCintersect!\n");
#endif
    return 1;
  }
#ifdef DEBUG
  printf("Returning 0 from offCCintersect!\n");
#endif
  return 0;
}

// this function takes a line defined by a starting and ending point
// and moves it a distance 'dist' to the right or the left side of
// the line. 'b' and 'e' are changed to the new offset line
void offsetLine(vect3* b, vect3* e, double dist, int side) {
  // side = 1 for left, side = -1 for right
  double x,y,xoffset,yoffset;

  x = e->x - b->x;
  y = e->y - b->y;

  // now offset the line a distance of 'dist' to side 'side'
  xoffset = dist*y/sqrt(x*x+y*y);
  yoffset = dist*x/sqrt(x*x+y*y);

  b->x += -side*xoffset;
  e->x += -side*xoffset;
  b->y +=  side*yoffset;
  e->y +=  side*yoffset;
}

// this function figures out where the center of the turning circle should
// be given that the path calls for doubling back on itself - i.e. a U turn
// this will preserve path length
void getUturncenter(vect3 sigma0, vect3 sigma1, double r, vect3*
c) {
  double a,b,qa,qb,qc;
  int hor,s;
  // dist is the distance from the double-back point
  // that the center of the turn circle needs to be
  // placed. We figured this out with the geometry of
  // three equal radius circles touching
```

```
  double dist = (7.0/6.0*PI - sqrt(3))*r;

  hor = getlinevals(sigma0,sigma1,&a,&b);

  if(hor)  // line equation is x = b;
  {
    c->x = sigma1.x;
    s = sign(sigma0.y-sigma1.y);
    c->y = sigma1.y + s*dist;
    return;
  }

  // line equation is y = ax + b
  // this is the solution to the set of equations:
  //                      y = ax + b
  //                      dist^2 = (x-sigma1.x)^2 + (y-sigma1.y)^2
  // put into quadratic variables
  qa = 1+a*a;
  qb = -2*(sigma1.x - a*b + a*sigma1.y);
  qc = sigma1.x*sigma1.x + sigma1.y*sigma1.y + b*b -2*b*sigma1.y - dist*dist;
  c->x = (-qb + sqrt(qb*qb - 4*qa*qc) ) / (2*qa);
  s = sign(sigma0.x - sigma1.x);
  if( s != sign(c->x - sigma1.x) )   // if we went the wrong way
    c->x = (-qb - sqrt(qb*qb - 4*qa*qc) ) / (2*qa);

  c->y = a*c->x + b;
}

void uavCopyCurrent(int id, uavInfo* K)  // copies g_uav_prev to
K. This so iterations in the algorithm {
// don't mess up g_uav_prev We use the previous g_uav information
  int i;                          // to match Matlab sequence and to avoid making a dependen
                                  // order of calls to rungkut and stateUpdate
  K->x = g_uav[id].x;
  K->y = g_uav[id].y;
  K->z = g_uav[id].z;
  K->psi = g_uav[id].psi;
  K->Vel = g_uav[id].Vel;
  K->turnFlag = g_uav[id].turnFlag;
  K->state = g_uav[id].state;
  K->newSigmaFlag = g_uav[id].newSigmaFlag;

  for(i=0;i<N;i++)
  {
    K->currentSigma[i].x = g_uav[id].currentSigma[i].x;
    K->currentSigma[i].y = g_uav[id].currentSigma[i].y;
```

```
    K->currentSigma[i].z = g_uav[id].currentSigma[i].z;
  }

  K->startNormal.x = g_uav[id].startNormal.x;
  K->startNormal.y = g_uav[id].startNormal.y;
  K->startNormal.z = g_uav[id].startNormal.z;
  K->endNormal.x = g_uav[id].endNormal.x;
  K->endNormal.y = g_uav[id].endNormal.y;
  K->endNormal.z = g_uav[id].endNormal.z;
}

int isLeftOfLine(vect3* p1, vect3* p2, double x, double y) {
  if( sign( (p2->x-p1->x)*(y-p1->y) - (p2->y-p1->y)*(x-p1->x)) == 1 )
    return 1;
  return 0;
}

// computes the 'delta' function on page 45 of Erik Anderson's thesis.
// This is the distance to complete half of a kappa turn. This is used
// to tell if a path segment is too short - i.e. if half the distance of
// the turn into the path segment plus half of the distance of the turn
// out of the path segment is longer than the path segment length, then
// the length of the segment is too short.
double halfKappaDist(double kappa, double beta, double r) {
  double dist;

  if(beta < 0.0001){              // u-turn
    return r*(7.0/6.0*PI - sqrt(3))/2.0;
  }

  if( fabs(beta - PI) < 0.0001){  // no turn
    return 0;
  }

  dist = 1 + sin(beta/2) + kappa*(1 - sin(beta/2));
  dist *= dist;
  dist = 4 - dist;

  if(dist < 0)
    dist = 0;
  else{
    dist = sqrt(dist);
    dist *= r;
  }

  dist += r*(1 + kappa*(1.0/sin(beta/2) - 1))*cos(beta/2);
```

```
  return dist;
} /*
 * File: traj_states.c
 * Created: Tue Dec 3
 *
 */

#define S_FUNCTION_NAME traj_states #define S_FUNCTION_LEVEL 2

/* N is the number of waypoints passed to trajGen */ #define N 3

#define NUM_INPUTS           1 #define INPUT_WIDTH        3*N+1
#define INPUT_FEEDTHROUGH    1

#define NUM_OUTPUTS          1 #define OUTPUT_WIDTH         14

#define NPARAMS              2

#define NUM_DISC_STATES      0 #define NUM_CONT_STATES      0

#define SAMPLE_TIME INHERITED_SAMPLE_TIME

#include "simstruc.h" #include "trajGen.h"

#define UAVDATA 0 #define UAVDATA_PARAM(S) ssGetSFcnParam(S,
UAVDATA)

#define INITWPS 1 #define INITWPS_PARAM(S) ssGetSFcnParam(S,
INITWPS)

void fillInParam(const mxArray* p, planeInfo* Param);

/*=====================*
 * S-function methods *
 *=====================*/

#define MDL_CHECK_PARAMETERS #if defined(MDL_CHECK_PARAMETERS) &&
defined(MATLAB_MEX_FILE) /* Function: mdlCheckParameters
================================================
 * Abstract:
 *    Validate our parameters to verify they are okay.
 */
static void mdlCheckParameters(SimStruct *S) {
  const mxArray *paramVal;
  mxArray *p;
```

```
double *pr;

paramVal = UAVDATA_PARAM(S);
if(!mxIsStruct(paramVal)){
  ssSetErrorStatus(S,"Parameter is not a struct!");
  return;
}

p = mxGetField(paramVal,0,"ic");
if(p == NULL ||
   !mxIsNumeric(p) ||
   !mxIsDouble(p) ||
   mxIsLogical(p) ||
   mxIsComplex(p) ||
   mxIsSparse(p) ||
   mxIsEmpty(p) ||
   mxGetN(p)*mxGetM(p) != 6 )
{
  ssSetErrorStatus(S,"wrong dimensions!");
  return;
}

p = mxGetField(paramVal,0,"psidot_max");
if(p == NULL                  ||
   !mxIsNumeric(p)            ||
   !mxIsDouble(p)             ||
   mxIsLogical(p)             ||
   mxIsComplex(p)             ||
   mxIsSparse(p)              ||
   mxIsEmpty(p)               ||
   mxGetN(p)*mxGetM(p) != 1 )
{
  ssSetErrorStatus(S,"Error in field 'psidot_max'!");
  return;
}

p = mxGetField(paramVal,0,"turn_param");
if(p == NULL                  ||
   !mxIsNumeric(p)            ||
   !mxIsDouble(p)             ||
   mxIsLogical(p)             ||
   mxIsComplex(p)             ||
   mxIsSparse(p)              ||
   mxIsEmpty(p)               ||
   mxGetN(p)*mxGetM(p) != 1 )
{
```

```
  ssSetErrorStatus(S,"Error in field 'turn_param'!");
  return;
}

p = mxGetField(paramVal,0,"equivdist");
if(p == NULL                    ||
   !mxIsNumeric(p)              ||
   !mxIsDouble(p)               ||
   mxIsLogical(p)               ||
   mxIsComplex(p)               ||
   mxIsSparse(p)                ||
   mxIsEmpty(p)                 ||
   mxGetN(p)*mxGetM(p) != 1 )
{
  ssSetErrorStatus(S,"Error in field 'equivdist'!");
  return;
}

p = mxGetField(paramVal,0,"kappa");
if(p == NULL                    ||
   !mxIsNumeric(p)              ||
   !mxIsDouble(p)               ||
   mxIsLogical(p)               ||
   mxIsComplex(p)               ||
   mxIsSparse(p)                ||
   mxIsEmpty(p)                 ||
   mxGetN(p)*mxGetM(p) != 1 )
{
  ssSetErrorStatus(S,"Error in field 'kappa'!");
  return;
}

paramVal = INITWPS_PARAM(S);
if(paramVal == NULL                    ||
   !mxIsNumeric(paramVal)              ||
   !mxIsDouble(paramVal)               ||
   mxIsLogical(paramVal)               ||
   mxIsComplex(paramVal)               ||
   mxIsSparse(paramVal)                ||
   mxIsEmpty(paramVal)                 ||
   mxGetN(paramVal)*mxGetM(paramVal) != 3*N )
{
  ssSetErrorStatus(S,"Error with parameter 'wps0'!");
  return;
}
```

```
}

#endif /* MDL_CHECK_PARAMETERS */

/* Function: mdlInitializeSizes
=============================================
 * Abstract:
 *   Setup sizes of the various vectors.
 */
static void mdlInitializeSizes(SimStruct *S) {
  ssSetNumSFcnParams(S, NPARAMS);  /* Number of expected parameters */
#if defined(MATLAB_MEX_FILE)
  if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
    mdlCheckParameters(S);
    if (ssGetErrorStatus(S) != NULL) {
      return;
    }
  } else {
    return; /* Parameter mismatch will be reported by Simulink */
  }
#endif

    ssSetNumContStates(S, NUM_CONT_STATES);
    ssSetNumDiscStates(S, NUM_DISC_STATES);

    if (!ssSetNumInputPorts(S, NUM_INPUTS)) return;
    ssSetInputPortWidth(S, 0, INPUT_WIDTH);
    ssSetInputPortRequiredContiguous(S,0,1);  /* so that we can access using
                          * 'ssGetInputPortRealSignal' */
    ssSetInputPortDirectFeedThrough(S, 0, INPUT_FEEDTHROUGH);

    if (!ssSetNumOutputPorts(S, NUM_OUTPUTS)) return;
    ssSetOutputPortWidth(S, 0, OUTPUT_WIDTH);

    ssSetNumSampleTimes(S, 1);  /*only one sample time */
    ssSetNumRWork(S, 2);        /*id for trajGen, previous time we calculated*/
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 1);        /*param struct for trajGen */
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    ssSetSFcnParamNotTunable(S,UAVDATA);  /* parameters are not tunable - i.e. wont
                        * change during simulation */
    ssSetSFcnParamNotTunable(S,INITWPS);  /* parameters are not tunable - i.e. wont
                        * change during simulation */
```

```
    /* Take care when specifying exception free code - see sfuntmpl_doc.c */
    /* ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE); */
    ssSetOptions(S, 0);
}


/* Function: mdlInitializeSampleTimes
=======================================
 * Abstract:
 *    Specifiy  the sample time.
 */
static void mdlInitializeSampleTimes(SimStruct *S) {
    ssSetSampleTime(S, 0, SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MDL_START #if defined(MDL_START)
  /* Function: mdlStart ============================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
static void mdlStart(SimStruct *S) {
  /* set up param */
  ssGetPWork(S)[0] = malloc(sizeof(planeInfo));
  fillInParam(UAVDATA_PARAM(S),(planeInfo*) (ssGetPWork(S)[0]));

  /* call trajGen initialize function */
  ssGetRWork(S)[0] = trajGenInitialize((planeInfo*) (ssGetPWork(S)[0]), mxGetPr(INITWPS_PAR
  ssGetRWork(S)[1] = 0.0;
} #endif /*  MDL_START */

static void mdlOutputs(SimStruct *S, int_T tid) {
  double psidotd,psid,vel, curTime,timestep;
  double statevars[12];
  int request_new_sigma;
  vect3 curPos,wpN[3];
  const real_T  *u  = ssGetInputPortRealSignal(S,0);
  real_T *y         = ssGetOutputPortRealSignal(S,0);
  planeInfo* Param = (planeInfo*) (ssGetPWork(S)[0]);
  int id = (int) (ssGetRWork(S)[0]);

  vel = u[0];
  wpN[0].x = u[1];
  wpN[0].y = u[2];
```

```
  wpN[0].z = u[3];
  wpN[1].x = u[4];
  wpN[1].y = u[5];
  wpN[1].z = u[6];
  wpN[2].x = u[7];
  wpN[2].y = u[8];
  wpN[2].z = u[9];

  curTime = ssGetT(S);
  timestep = curTime-ssGetRWork(S)[1];
  ssGetRWork(S)[1] = curTime;

  /* call trajGen update and output functions */
  request_new_sigma = trajGenOutput(id,vel,statevars, timestep,Param);
  curPos.x = statevars[0];
  curPos.y = statevars[1];
  psid     = statevars[2];
  psidotd  = statevars[6];

  trajGenUpdate(id,vel,wpN,Param,timestep);

  y[0]  = (double) request_new_sigma;
  y[1]  = Param->ic.z0;
  y[2]  = curPos.x;
  y[3]  = curPos.y;
  y[4]  = psid;
  y[5]  = vel;
  y[6]  = statevars[4];   //xdotd
  y[7]  = statevars[5];   //ydotd
  y[8]  = statevars[6];   //psidotd
  y[9]  = statevars[7];   //vdotd
  y[10] = statevars[8];   //xdotdotd
  y[11] = statevars[9];   //ydotdotd
  y[12] = statevars[10];  //psidotdotd
  y[13] = statevars[11];  //vdotdotd
}

/* Function: mdlTerminate
=======================================================
 */
static void mdlTerminate(SimStruct *S) {
  /* call trajGen destroy function */
  trajGenDestroy();
  free(ssGetPWork(S)[0]);
}
```

```
void fillInParam(const mxArray* p, planeInfo* Param) {
  mxArray* f = mxGetField(p,0,"ic");
  double* ic = mxGetPr(f);
  Param->ic.x0 = ic[0];
  Param->ic.y0 = ic[1];
  Param->ic.z0 = ic[4];
  Param->ic.psi0 = ic[2];
  Param->ic.zdot0 = ic[5];
  Param->ic.v0 = ic[3];

  f = mxGetField(p,0,"psidot_max");
  Param->psidot_max = mxGetScalar(f);

  f = mxGetField(p,0,"turn_param");
  Param->turn_param = (int) mxGetScalar(f);

  f = mxGetField(p,0,"equivdist");
  Param->equivdist = (int) mxGetScalar(f);

  f = mxGetField(p,0,"kappa");
  Param->kappa = mxGetScalar(f);

  Param->k1 = 10.0;
  Param->k2 = 1.3;
}

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a
MEX-file? */ #include "simulink.c"      /* MEX-file interface
mechanism */ #else #include "cg_sfun.h"       /* Code generation
registration function */ #endif


/*----------------------------------------*/

/* Name: cmnStructs.h                     */

/* Created: 8/5/2002                      */

/*----------------------------------------*/



/** This file contains all the common structures

 *  to be used in sfunction.c
```

```
 */


#ifndef CMNSTRUCTS_H

#define CMNSTRUCTS_H


#include <list>

#include <math.h>


#ifndef FOR_WINDOWS

#include <pthread.h>                          /**< used for threads
which are used in the openGL engine */

#endif


#define N 3                                   /**< N is the number
of waypoints the traj_gen sfunction receives */



typedef struct {


  double x;                                   /**< x coordinate of vector */

  double y;                                   /**< y coordinate of vector */

  double z;                                   /**< z coordinate of vector */


} vect3;
```

```
typedef struct {

  double x;
  double y;
  double r;
  std::list<vect3> nodes;

} ThreatStruct;

typedef struct {

  double x0;                    /**< initial inertial position x */
  double y0;                    /**< initial inertial position y */
  double z0;                    /**< initial altitude */
  double psi0;                  /**< initial heading */
  double zdot0;                 /**< initial climb rate */
  double v0;                    /**< initial velocity */

} ic_t;

typedef struct {
```

```
  ic_t   ic;                    /**< initial conditions */
  double psidot_max;            /**< turning rate limit */
  int    turn_param;            /**< 1=parameterized turn in */
                                /**< trajectory generator */
  int    equivdist;             /**< 1=trajectories have same */
                                /**< length as Voronoi path */
  double kappa;                 /**< parameter for trajectory */
                                /**< generator */

  double k1;                    /**< k1 used in the control law p. 47 of thes
  double k2;                    /**< k2 damping factor in control law */

} planeInfo;

typedef struct{
  double th_x;
  double th_y;
  double xd;
  double yd;
  double psid;
  double vd;
  double psidotd;
} attInfo;
```

```
/*******Global Defines for the entire Program ****************/

#define PI 3.141592653589793                    /**< pi - could be
better precision */

#define TIME_STEP 0.0005                        /**< global time
step */

//#define TIME_STEP 0.01

#define SEND_TIME 0.05

#define MAXPLANES 10


/** All memory allocations are freed for the program and for each
individual

 *  sfunction also any file pointers are closed. This prototype is in here so that

 *  every file can have access to this function

 */

void simUavDestroy();

#endif

/*----------------------------------------*/ /* Name: trajGen.h
*/ /* Created: 8/5/2002                     */
/*----------------------------------------*/

/** This file contains all the function prototypes and globals
 *  to be used in sfunction.c
 */

#ifndef TRAJGEN_H #define TRAJGEN_H

#include <stdio.h> #include "cmnStructs.h" #include <math.h>

/*********** UAV info struct *****************/ typedef struct {

  double x;                                     /**< current x position of the plane */
```

```
  double y;                                  /**< current y position of the plane */
  double z;                                  /**< current z position of the plane */
  double Vel;                                /**< current airspeed V */
  double psi;                                /**< current heading of the plane */

  int turnFlag;                              /**< turn initiaed flag
                        *   0 for not turning
                        *   1 for turning
                        *   2 for just finished a turn
                        *   3 for turning to a path normal (when far away from path)
                        *   4 for tracking normal
                        *   5 for turning from normal to path
                        *   6 for first segment of hit-waypoint turn
                        *   7 for second segment of hit-waypoint turn
                        *   8 for third segment of hit-waypoint turn
                        */

  int state;                                  /**< current state of state machine */
  vect3 currentSigma[N];                      /**< current value of sigma */
  int newSigmaFlag;                           /**< request for new sigma */
  vect3 startNormal;                          /**< starting coordinate of the normal vector
  vect3 endNormal;                            /**< ending coordinate of the normal vector *
} uavInfo;

/** Initializes all the structs to be used in the sfunction */ int
trajGenInitialize(planeInfo*, double*);

/** Destorys any memory allocated by the sfunction and closes any
file pointers */ void trajGenDestroy();

/** Determines the output of the sfunction */ int
trajGenOutput(int, double, double*,double,planeInfo*);

/** Calls both the state update and the Runge-Kutta method */ void
trajGenUpdate(int, double, vect3*,planeInfo*,double);

/** Updates the discreet variables used in the sfucntion */ void
stateUpdate(uavInfo*, double, vect3*, planeInfo*);

/** Derivative used in the s-function */ void
trajGenDerivative(uavInfo*, double, planeInfo*);

/** 4th order runge-kutta method for determining trajectory
generation */ void rungkutTrajGen(int, double, planeInfo*,double);

/** To copy g_uav to compute psidotd */ void
```

```
uavCopyCurrent(int,uavInfo*);

/** returns the values of the line a,b in y = ax + b, and hor is
set to 0
 *   if the line is x = b, then hor is set to 1 and b is returned
 */
int getlinevals(vect3,vect3,double*,double*);

/** turn specifies the direction of the turn.  turn is positive
for a ccw
 *   turn and negative for a cw turn.  0 is returned if the next path requires
 *   no turn.  Currently, we are on the path specified by q1.  The path we
 *   are switching to is q2.
 */
int turndir(vect3,vect3);

/** returns the sign of a number, with 0 having a sign of 0 */ int
sign(double);

/** return the norm of a vector - currently only in 2 dimensions
(x and y) */ double norm(vect3);

/** p is the starting waypoint of the path.  q is a unit vector in
the
 *   direction of the path.  The Euclidean distance between the path and the
 *   airplane is returned.
 */
double dist2path(vect3,vect3,double,double);

/**
 *This function returns a value for psidot for tracking a path seqment
 *   when we are near the path.
 *   p is the beginning waypoint of our path.  q is a unit vector in the
 *   direction of the path.  eps_d is the maximum distance from the path
 *   for which we say we are tracking the line.
 */
double
track(vect3,vect3,double,double,double,double,double,double,planeInfo*);

/*
 *   returns ci = true if either circle intersects the next path segment
 *   and ci = false if neither circle intersects the next path segment.
 *   hor,a,b are the values returned from getlinevals
 */
int
clintersect(int,double,double,double,double,double,double,double);
```

```
/** this takes the norm of a N point waypoint path
 *  basically it's just to see if the path has changed */
double normN(vect3 s[], vect3 t[]);

/** copys a set of N waypoints to another N waypoint path */ void
sigmaCopy(vect3 fromS[], vect3 toS[]);

// returns the angle between the path formed by the next
// three waypoints in the order w0, w1, w2
double getBeta(vect3* w0, vect3* w1, vect3* w2);

/** Returns the value of kappa that makes the length of the path
segment and
 *  the length of the constrained turn equal to within 2^(-n) where n is a
 *  positive integer.
 */
double GetKappa(double,double,int);

/** Returns the value of the distance between the path segment and
the
 *  constrained turn for the angle beta between the successive path segments,
 *  the value k (i.e. kappa which lies between 0 and 1), and the turning
 *  radius r
 */
double Lambda(double,double,double);

/** Returns the center of the cicle so the plane passes through a
waypoint.
 *  p is the waypoint to pass through, q1 is a unit vector in the direction
 *  of the current path segment, q2 is a unit vector in the direction of
 *  the next path segment, and radius is the radius of the turn.
 *  Assume that q1 and q2 are not parallel.
 */
void getturncenter(vect3,vect3,vect3,double,double,double,vect3*);

/** determines whether two circles intersect.  1 is returned if
they do.
 *  Circle 1 has center c1 and radius r1.  Similarly for circle 2.
 */
int ccintersect(uavInfo*,vect3,double,vect3,double);

// changes a vector to a unit vector
void makeUnitVector(vect3*);

// for a vector with beginning point 'b' and ending point 'e' and a point 'p'
```

```
// this function returns -1 for 'p' on the right of the vector, 1 for 'p' on
// the left of the vector and 0 for 'p' on the vector
int getSideOfLine(vect3* b, vect3* e, vect3* p);

// if the uav is flying too far off the path to intersect the circle
// then this function is called. It calculates the point where the
// circles should intersect, then it creates a line from that point
// perpendicular to the current path segment and checks to see if the
// circle has crossed the line. If it has it returns true
int offPathCCintersect(uavInfo*,vect3 c, double r, vect3
sideCircle);

// this function takes a line defined by a starting and ending point
// and moves it a distance 'dist' to the right or the left side of
// the line. 'b' and 'e' are changed to the new offset line
void offsetLine(vect3* b, vect3* e, double dist, int side);

// this function figures out where the center of the turning circle should
// be given that the path calls for doubling back on itself - i.e. a U turn
// this will preserve path length
void getUturncenter(vect3 sigma0, vect3 sigma1, double r, vect3*
c);

// computes the 'delta' function on page 45 of Erik Anderson's thesis.
// This is the distance to complete half of a kappa turn. This is used
// to tell if a path segment is too short - i.e. if half the distance of
// the turn into the path segment plus half of the distance of the turn
// out of the path segment is longer than the path segment length, then
// the length of the segment is too short.
double halfKappaDist(double kappa, double beta, double r);

#endif
```

10.3 Other files

Supporting DLL files are on the CD accompanying the report.